

MATLAB[®]/Simulink[®] for Digital Communication

Won Y. Yang, Yong S. Cho, Won G. Jeon, Jeong W. Lee, Jong H. Paik
Jae K. Kim, Mi-Hyun Lee, Kyu I. Lee, Kyung W. Park, Kyung S. Woo

Copyright © 2009 by A-Jin Publishing Co

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher. Requests for permission or further information should be addressed to the Publisher, A-Jin.

Limits of Liability and Disclaimer of Warranty of Software

The authors and publisher of this book have used their best efforts and knowledge in preparing this book as well as developing the computer programs in it. However, they make no warranty of any kind, expressed or implied, with regard to the programs or the documentation contained in this book. Accordingly, they shall not be liable for any incidental or consequential damages in connection with, or arising out of, the readers' use of, or reliance upon, the material in this book.

The reader is expressly warned to consider and adopt all safety precautions that might be indicated by the activities herein and to avoid all potential hazards. By following the instructions contained herein, the reader willingly assumes all risks in connection with such instructions.

MATLAB[®] and Simulink[®] are registered trademarks of The MathWorks, Inc. and are used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB[®] and Simulink[®] does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB[®] and Simulink[®]. For MATLAB[®] and Simulink[®] product information, please contact:

The MathWorks, Inc.

3 Apple Hill Drive

Natick, MA 01760-2098, USA

☎: 508-647-7000, Fax: 508-647-7001

E-mail: info@mathworks.com

Web: www.mathworks.com

Questions about the contents of this book can be mailed to wyyang53@hanmail.net.

Program files in this book can be down-loaded from the following website:

<http://wyyang53.com.ne.kr/>

ISBN 0

Printed in Korea by A-Jin Publishing Co., Korea

*To our parents and families
who love and support us
and
to our teachers and students
who enriched our knowledge*

Table of Contents

PREFACE	iii
CHAPTER 1: FOURIER ANALYSIS	1
1.1 CONTINUOUS-TIME FOURIER SERIES (CTFS)	2
1.2 PROPERTIES OF CTFS	6
1.2.1 Time-Shifting Property	6
1.2.2 Frequency-Shifting Property	6
1.2.3 Modulation Property	6
1.3 CONTINUOUS-TIME FOURIER TRANSFORM (CTFT)	7
1.4 PROPERTIES OF CTFT	13
1.4.1 Linearity	13
1.4.2 Conjugate Symmetry	13
1.4.3 Real Translation (Time Shifting) and Complex Translation (Frequency Shifting)	14
1.4.4 Real Convolution and Correlation	14
1.4.5 Complex Convolution – Modulation/Windowing	14
1.4.6 Duality	17
1.4.7 Parseval Relation - Power Theorem	18
1.5 DISCRETE-TIME FOURIER TRANSFORM (DTFT)	18
1.6 DISCRETE-TIME FOURIER SERIES - DFS/DFT	19
1.7 SAMPLING THEOREM	21
1.7.1 Relationship between CTFS and DFS	21
1.7.2 Relationship between CTFT and DTFT	27
1.7.3 Sampling Theorem	27
1.8 POWER, ENERGY, AND CORRELATION	29
1.9 LOWPASS EQUIVALENT OF BANDPASS SIGNALS	30
Problems	36
CHAPTER 2: PROBABILITY AND RANDOM PROCESSES	39
2.1 PROBABILITY	39
2.1.1 Definition of Probability	39
2.1.2 Joint Probability and Conditional Probability	40
2.1.3 Probability Distribution/Density Function	41
2.1.4 Joint Probability Density Function	41
2.1.5 Conditional Probability Density Function	41
2.1.6 Independence	41
2.1.7 Function of a Random Variable	42
2.1.8 Expectation, Covariance, and Correlation	43
2.1.9 Conditional Expectation	47
2.1.10 Central Limit Theorem - Normal Convergence Theorem	47
2.1.11 Random Processes	49
2.1.12 Stationary Processes and Ergodic Processes	51
2.1.13 Power Spectral Density (PSD)	53
2.1.14 White Noise and Colored Noise	53
2.2 LINEAR FILTERING AND PSD OF A RANDOM PROCESS	57
2.3 FADING EFFECT OF A MULTI-PATH CHANNEL	59
Problems	62

CHAPTER 3: ANALOG MODULATION	71
3.1 AMPLITUDE MODULATION (AM).....	71
3.1.1 DSB (Double Sideband)-AM (Amplitude Modulation)	71
3.1.2 Conventional AM (Amplitude Modulation)	75
3.1.3 SSB (Single Sideband)-AM(Amplitude Modulation).....	78
3.2 ANGLE MODULATION - FREQUENCY/PHASE MODULATIONS	82
Problems	86
CHAPTER 4: ANALOG-TO-DIGITAL CONVERSION	87
4.1 QUANTIZATION.....	87
4.1.1 Uniform Quantization	88
4.1.2 Non-uniform Quantization	89
4.1.3 Non-uniform Quantization Considering Relative Errors	91
4.2 Pulse Code Modulation (PCM).....	95
4.3 Differential Pulse Code Modulation (DPCM)	97
4.4 Delta Modulation (DM)	100
Problems	103
CHAPTER 5: BASEBAND DIGITAL TRANSMISSION	107
5.1 RECEIVER (RCVR) and SNR	107
5.1.1 Receiver of RC Filter Type	109
5.1.2 Receiver of Matched Filter Type	110
5.1.3 Signal Correlator	112
5.2 SIGNALING AND ERROR PROBABILITY.....	114
5.2.1 Antipodal (Bipolar) Signaling.....	114
5.2.2 OOK(On-Off Keying)/Unipolar Signaling	118
5.2.3 Orthogonal Signaling	119
5.2.4 Signal Constellation Diagram	121
5.2.5 Simulation of Binary Communication	123
5.2.6 Multi-level(amplitude) PAM Signaling	127
5.2.7 Multi-dimensional Signaling.....	129
5.2.8 Bi-orthogonal Signaling	133
Problems	136
CHAPTER 6: BANDLIMITED CHANNEL AND EQUALIZER	139
6.1 BANDLIMITED CHANNEL	139
6.1.1 Nyquist Bandwidth.....	139
6.1.2 Raised-Cosine Frequency Response	141
6.1.3 Partial Response Signaling - Duobinary Signaling	143
6.2 EQUALIZER.....	148
6.2.1 Zero-Forcing Equalizer (ZFE)	148
6.2.2 MMSE Equalizer (MMSEE).....	151
6.2.3 Adaptive Equalizer (ADE).....	154
6.2.4 Decision Feedback Equalizer (DFE).....	155
Problems	159
CHAPTER 7: BANDPASS DIGITAL TRANSMISSION	169
7.1 AMPLITUDE MODULATION - AMPLITUDE SHIFT KEYING (ASK).....	169
7.2 FREQUENCY MODULATION - FREQUENCY SHIFT KEYING (FSK).....	178
7.3 PHASE MODULATION - PHASE SHIFT KEYING (PSK)	187
7.4 DIFFERENTIAL PHASE SHFT KEYING (DPSK).....	190
7.5 QUADRATURE AMPLITUDE MODULATION (QAM) - PAM/PSK	195

7.6 COMPARISON OF VARIOUS SIGNALINGS.....	200
Problems	205
CHAPTER 8: CARRIER RECOVERY AND SYMBOL SYNCHRONIZATION	225
8.1 INTRODUCTION.....	225
8.2 PLL (PHASE-LOCKED LOOP)	226
8.3 ESTIMATION OF CARRIER PHASE USING PLL	231
8.4 CARRIER PHASE RECOVERY	233
8.4.1 Carrier Phase Recovery Using a Squaring Loop for BPSK Signals	233
8.4.2 Carrier Phase Recovery Using Costas Loop for PSK Signals	235
8.4.3 Carrier Phase Recovery for QAM Signals	238
8.5 SYMBOL SYNCHRONIZATION (TIMING RECOVERY)	241
8.5.1 Early-Late Gate Timing Recovery for BPSK Signals	241
8.5.2 NDA-ELD Synchronizer for PSK Signals	244
Problems	247
CHAPTER 9: INFORMATION AND CODING	255
9.1 MEASURE OF INFORMATION - ENTROPY	255
9.2 SOURCE CODING.....	256
9.2.1 Huffman Coding.....	256
9.2.2 Lempel-Zip-Welch Coding	259
9.2.3 Source Coding vs. Channel Coding	262
9.3 CHANNEL MODEL AND CHANNEL CAPACITY	263
9.4 CHANNEL CODING	268
9.4.1 Waveform Coding	269
9.4.2 Linear Block Coding	270
9.4.3 Cyclic Coding	279
9.4.4 Convolutional Coding and Viterbi Decoding	284
9.4.5 Trellis-Coded Modulation (TCM).....	293
9.4.6 Turbo Coding	297
9.4.7 Low-Density Parity-Check (LDPC) Coding	308
9.4.8 Differential Space-Time Block Coding (DSTBC).....	313
9.5 CODING GAIN	316
Problems	318
CHAPTER 10: SPREAD-SPECTRUM SYSTEM	337
10.1 PN (Pseudo Noise) Sequence	337
10.2 DS-SS (Direct Sequence Spread Spectrum)	345
10.3 FH-SS (Frequency Hopping Spread Spectrum).....	350
Problems	354
CHAPTER 11: OFDM SYSTEM	357
11.1 OVERVIEW OF OFDM.....	357
11.2 FREQUENCY BAND AND BANDWIDTH EFFICIENCY OF OFDM	361
11.3 CARRIER RECOVERY AND SYMBOL SYNCHRONIZATION	362
11.4 CHANNEL ESTIMATION AND EQUALIZATION	379
11.5 INTERLEAVING AND DEINTERLEAVING.....	382
11.6 PUNCTURING AND DEPUNCTURING	384
11.7 IEEE STANDARD 802.11A - 1999	386
Problems	393

APPENDICIES	407
Appendix A: Fourier Series/Transform	407
Appendix B: Laplace Transform and z -Transform	412
Appendix C: Differentiation w.r.t. a Vector	414
Appendix D: Useful Formulas	415
Appendix E: MATLAB Introduction.....	417
Appendix F: Simulink	421
REFERENCES	425
INDEX	427

Preface

This book has been designed as a reference book for students or engineers studying communication systems possibly in the curriculum of Electrical Engineering program rather than a text book for any course on communication. Readers are supposed to have taken at least two junior-level courses, one on signals and systems and another one on probability and random processes. In other words, readers should have a basic knowledge about the linear system, Fourier transform, Laplace transform, z -transform, probability, and random processes although the first two chapters of this book provide a brief overview of some background topics to minimize the necessity of the prerequisite courses and to refresh their memory if nothing else.

It is not the aim of this book to provide any foundation in the basic theory of digital communication since the authors do not have such a deep knowledge as to do it. The first aim of this book is to help the readers understand the concepts, techniques, terminologies, equations, and block diagrams appearing in the existing books on communication systems while using MATLAB® to simulate the various communication systems most of which are described by block diagrams and equations. Needless to say, the readers are recommended to learn some basic usage of MATLAB® that is available from the MATLAB help function or the on-line documents at the web site <<http://www.mathworks.com/matlabcentral/>>. However, they are not required to be so good at MATLAB® since most programs in this book have been composed carefully and completely so that they can be understood in connection with related/referred equations and/or block diagrams. The readers are expected to get used to MATLAB software while trying to modify/use the MATLAB® codes and Simulink® models in this book for solving the end-of-chapter problems or their own problems. The second and main aim of this book is to make even a novice at both MATLAB® and communication systems become acquainted, at least comfortable, with MATLAB® as well as communication systems while running the MATLAB programs on his/her computer and trying to understand what is going on in the systems simulated by the programs. Is it too much to expect that a novice will become interested in communications and simultaneously fall in love with MATLAB®, which is a universal language for engineers and scientists after having read this book through? Is it just the authors' imagination that the readers would think of this book describing and explaining many concepts in MATLAB® rather than in English? In any case, the authors have no intention to hide their hope that this book will be one of the all-the-time-reserved books in most libraries and can be found always on the desks of most communication engineers. The features of this book can be summarized as follows:

1. This book presents more MATLAB programs for the simulation of communication systems than any existent books with the same or similar titles as an approach to explain most things using MATLAB® and figures rather than English and equations.
2. Most MATLAB programs are presented in a complete form so that the readers can run them instantly with no programming skill and focus on understanding the behavior and characteristic of the simulated systems and making interpretations based on the tentative and final simulation results.
3. Many programs have a style of on-line processing rather than batch processing so that the readers can easily understand the whole system and the underlying algorithm in details block by block and operation by operation. Furthermore, the on-line processing style of the programs is expected to let the readers develop their insight into the real system.
4. Authors never think that this book can replace the existent books made by many great authors to whom they are not comparable to. They neither expect that this book can take the place of the MATLAB manual. Instead, this book is designed to play a role of bridge

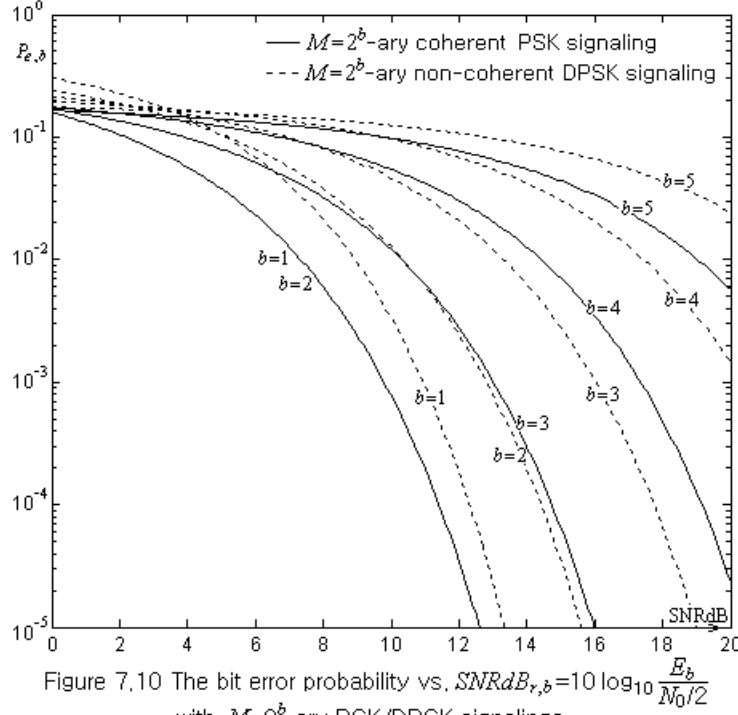
between MATLAB[®] software and the theory, block diagrams, and equations appearing in the field of communications so that the readers can feel free to utilize MATLAB[®] software for studying communication systems and become much more interested in communications than before reading this book.

The contents of this book are derived from the works of many (known or unknown) great scientists, scholars, and researchers, all of whom are deeply appreciated. We would like to thank the reviewers for their valuable comments and suggestions, which contribute to enriching this book.

We also thank the people of the School of Electronic & Electrical Engineering, Chung-Ang University for giving us an academic environment. Without affections and supports of our families and friends, this book could not be written. Special thanks should be given to Senior Researcher Yong-Suk Park for his invaluable help in correction. We gratefully acknowledge the editorial and production staff of A-Jin Publishing Company for their kind, efficient, and encouraging guide.

Program files can be downloaded from <<http://wyyang53.com.ne.kr/>>. Any questions, comments, and suggestions regarding this book are welcome and they should be mailed to wyyang53@hanmail.net.

Won Young Yang et al.



7.5 QUADRATURE AMPLITUDE MODULATION (QAM)

The passband $M=2^b$ -ary QAM signaling uses the waveforms which have different amplitudes and phases depending on what data they are carrying and therefore, it can be viewed as a kind of APK (amplitude-phase keying), which combines amplitude modulation and phase modulation. Each of the passband $M=2^b$ -ary QAM signal waveforms can be written as

$$\begin{aligned}
 s_m(t) &= A_{mc}s_{uc}(t) + A_{ms}s_{us}(t) = \text{Re} \left\{ (A_{mc} + jA_{ms}) \sqrt{\frac{2}{T_s}} e^{j\omega_c t} \right\} \quad \text{for } m=0, 1, \dots, M-1 \\
 &= A_{mc} \sqrt{\frac{2}{T_s}} \cos(\omega_c t) - A_{ms} \sqrt{\frac{2}{T_s}} \sin(\omega_c t) \quad \text{for } 0 \leq t < T_s \\
 &= A_m \sqrt{\frac{2}{T_s}} \cos(\omega_c t + \theta_m) \quad \text{with } A_m = \sqrt{A_{mc}^2 + A_{ms}^2} \text{ and } \theta_m = \tan^{-1} \frac{A_{ms}}{A_{mc}}
 \end{aligned} \tag{7.5.1}$$

where

$$s_{uc}(t) = \sqrt{\frac{2}{T_s}} \cos(\omega_c t), \quad s_{us}(t) = -\sqrt{\frac{2}{T_s}} \sin(\omega_c t) : \text{Basis signal waveforms} \tag{7.5.2}$$

T_b : Bit time or bit duration, $T_s = bT_b$: Symbol time or symbol duration

$E_s = bE_b$: Signal energy per symbol

The QAM signal waveforms are illustrated in Fig. 7.1(a4) and each of them can be represented as a vector of length A_m

$$\mathbf{s}_m = [A_{mc} \ A_{ms}] = A_m [\cos \theta_m \ \sin \theta_m] \quad (7.5.3)$$

and depicted in the signal space as Fig. 7.1(b4) or Fig. 7.11 where the (orthonormal) bases of the signal space are the unit vectors representing $s_{uc}(t)$ and $s_{us}(t)$ defined by Eq. (7.5.2).

Suppose the amplitudes/phases of the $M=2^b$ -ary QAM signal waveforms are designed in such a way that they can be represented by a rectangular constellation in the signal space as Fig. 7.11(a) and the minimum distance among the signal points is $2A$. Then, the average $E_{s,av}$ of signal powers (represented by the squared distance between signal points and the origin) and the average number N_b of adjacent signal points for a signal point vary with the modulation order $M=2^b$ or the number b of bits per symbol as

$$M = 2^2 = 4: E_{s,av} = \frac{1}{M} \sum_{m=1}^{M-1} A_m^2 = \frac{4A^2 \times (1^2) \times 2}{4} = 2A^2 = \frac{2(M-1)}{3} A^2, \quad N_b = 2 = 4 - \frac{4}{\sqrt{M}=2}$$

$$M = 2^4 = 16: E_{s,av} = \frac{1}{M} \sum_{m=1}^{M-1} A_m^2 = \frac{4A^2 \times (1^2 + 3^2) \times 4}{16} = 10A^2 = \frac{2(M-1)}{3} A^2$$

$$N_b = \frac{4 \times ((2-1)^2 \times 4 + (4-2) \times 3 + 2)}{4^2} = 3 = 4 - \frac{4}{\sqrt{M}=4}$$

$$M = 4 \times 2^4 = 64: E_{s,av} = \frac{4A^2 \times (1^2 + 3^2 + 5^2 + 7^2) \times 8}{M=64} = 42A^2 = \frac{2(M-1)}{3} A^2$$

$$N_b = \frac{4 \times ((4-1)^2 \times 4 + (8-2) \times 3 + 2)}{8^2} = \frac{7}{2} = 4 - \frac{4}{\sqrt{M}=8}$$

.....

$$E_{s,av} = \frac{2(M-1)}{3} A^2: \text{Average signal energy per symbol} \quad (7.5.4a)$$

$$N_b = 4 - \frac{4}{\sqrt{M}}: \text{Average number of adjacent signal points} \quad (7.5.4b)$$

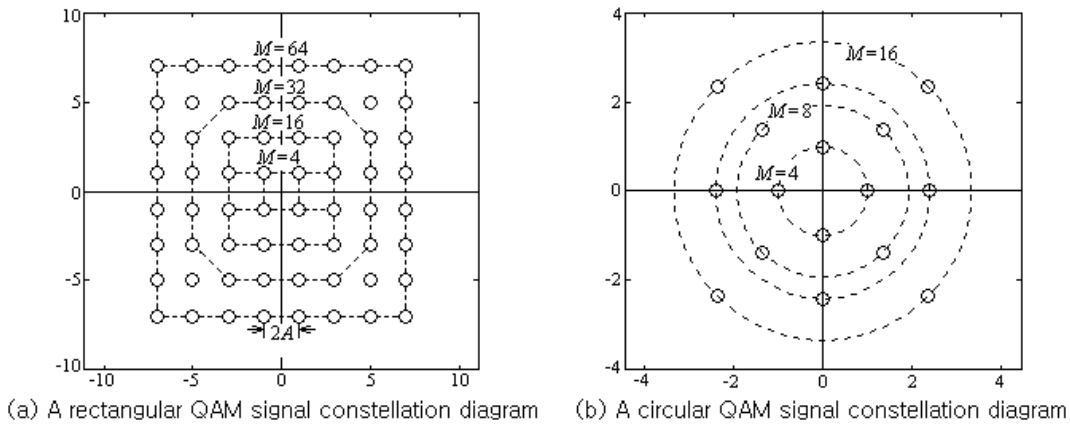


Figure 7.11 QAM signal constellation diagrams

A half of the minimum distance d_{\min} among the $M=2^b$ -ary QAM signal points in the signal space can be expressed in terms of the average signal energy $E_{b,av}$ per bit as

$$\frac{d_{\min}}{2} = A \stackrel{(7.5.4a)}{=} \sqrt{\frac{3/2}{M-1} E_{s,av}} = \sqrt{\frac{3/2}{M-1} b E_{b,av}} \quad (7.5.5)$$

Note that if we use the circular constellation as in Fig. 7.11(b), we may have a larger minimum distance with the same average energy $E_{b,av}$, but the difference is very small for $M \geq 16$. Besides, an $M=2^b$ ($b=2m$: an even number) -ary QAM signaling with rectangular signal constellation can easily be implemented by two independent $2^{b/2}$ -ary PAM signaling, each of which uses one of the quadrature carriers $\cos(\omega_c t)$ and $\sin(\omega_c t)$, respectively (see Fig. 7.12). This is why QAM signaling with rectangular signal constellation is widely used.

For an $M=2^b = LN$ -ary QAM signaling implemented by combining an L -ary PAM signaling and an N -ary PAM signaling, the symbol error probability can be found as

$$P_{e,s}(M=LN) = 1 - P(\text{probability of correct detection}) = 1 - (1 - P_{e,s}(L))(1 - P_{e,s}(N))$$

$$\stackrel{(7.1.5)}{=} 1 - \left[1 - \frac{2(L-1)}{L} Q\left(\sqrt{\frac{3b/2}{L^2-1} SNR_{r,b}}\right) \right] \left[1 - \frac{2(N-1)}{N} Q\left(\sqrt{\frac{3b/2}{N^2-1} SNR_{r,b}}\right) \right] \quad (7.5.6)$$

$$\leq \frac{4(L-1)}{L} Q\left(\sqrt{\frac{3b/2}{M-1} SNR_{r,b}}\right) \text{ with } L \geq N \quad (7.5.7)$$

where the upperbound on the RHS coincides with what is obtained by substituting Eqs. (7.5.4b) and (7.5.5) into Eq. (5.2.41). How about the bit error probability? Under the assumption that the information symbols are Gray-coded so that the codes for adjacent signal points differ in only one bit, the most frequent symbol errors contain just one of the b bits mistaken and the relationship between the symbol and bit errors can be written as

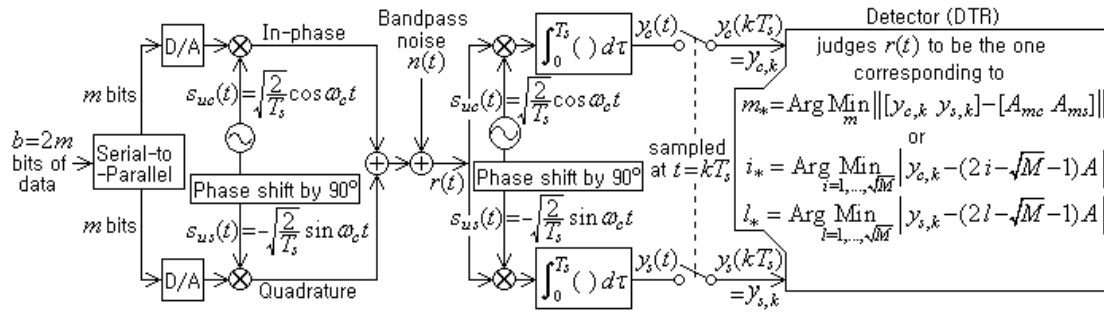
$$P_{e,b} = \frac{1}{b} P_{e,s} \quad (7.5.8)$$

Now, let us think about the structure of the $M=2^b = 2^{2m}$ ($b=2m$: an even number) -ary QAM communication system depicted in Fig. 7.12 where the XMTR divides the b bits of a message symbol data into two parts of $m=b/2$ bits, converts them to analog signals, and modulates them with the *quadrature carriers* that are the basis signal waveforms

$$s_{uc}(t) = \sqrt{\frac{2}{T_s}} \cos(\omega_c t) \quad \text{and} \quad s_{us}(t) = -\sqrt{\frac{2}{T_s}} \sin(\omega_c t), \quad (7.5.10)$$

respectively. This QAM scheme is basically equivalent to performing two independent quadrature PAMs in parallel. The RCVR has two *quadrature correlators*, each of which computes a correlation of the received signal $r(t)$ with $s_{uc}(t)$ and $s_{us}(t)$ to make the sampled outputs

$$y_{c,k} \stackrel{(5.1.27)}{=} \int_0^{T_s} s_{uc}(t) r(t + (k-1)T_s) dt \quad \text{and} \quad y_{s,k} \stackrel{(5.1.27)}{=} \int_0^{T_s} s_{us}(t) r(t + (k-1)T_s) dt, \quad (7.5.11)$$

Figure 7.12 The structure of an $M=2^b=2^{2m}=L^2$ -ary QAM communication system

respectively. The DTR judges the received signal to be the one represented by the signal point which is the closest to the point $(y_{c,k}, y_{s,k})$ in the signal space as

$$m_* = \text{Arg Min}_m \|[y_{c,k} \ y_{s,k}] - [A_{mc} \ A_{ms}]\| = \text{Arg Min}_m \{(y_{c,k} - A_{mc})^2 + (y_{s,k} - A_{ms})^2\} \quad (7.5.12)$$

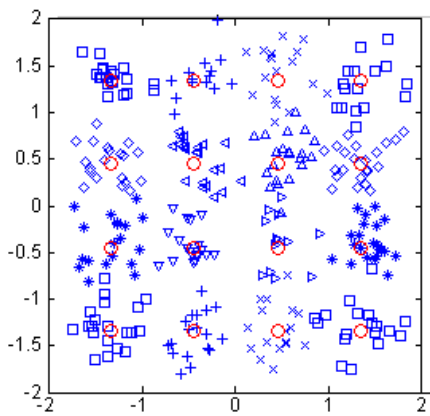
or combines two independent quadrature PAM demodulation results

$$i_* = \text{Arg Min}_{i=1,\dots,\sqrt{M}} |y_{c,k} - (2i - \sqrt{M} - 1)A| \quad (7.5.13a)$$

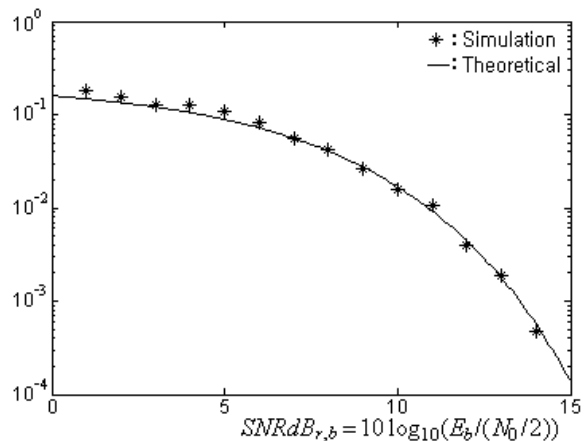
$$l_* = \text{Arg Min}_{l=1,\dots,\sqrt{M}} |y_{s,k} - (2l - \sqrt{M} - 1)A| \quad (7.5.13b)$$

to judge the received signal to be the one represented by the $(i_*, l_*)^{\text{th}}$ signal point from the left-lower corner in the signal space.

The objective of the following MATLAB program “sim_QAM_passband.m” is to simulate the passband $M=2^b=2^4$ -ary QAM signaling depicted in Fig. 7.12 and plot the bit error probability vs. $\text{SNRdB}_{r,b} = 10 \log_{10}(E_b/(N_0/2))$ for checking the validity of theoretical derivation results (7.5.8).



(a) A QAM signal constellation diagram



(b) The BER (bit error rate) curve - Bit error probability

Figure 7.13 The signal constellation diagram and BER curve for an $M=2^b=2^{2m}=L^2$ -ary QAM signaling

```

%sim_QAM_passband.m
% simulates a digital communication system in Fig.7.13
% with QAM signal waveforms in Fig.7.11
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
clear, clf
b=4; M=2^b; L=2^(b/2); % # of bits per symbol and the modulation order
SNRdbt=0:0.1:15; SNRbt=10.^(SNRdbt/10);
Pm=2*(1-1/L)*Q(sqrt(3/2*b*SNRbt/(M-1))); % Eq.(7.1.5)
pobet= (1-(1-Pm).^2)/b; % Eq.(7.5.8) with (7.5.6)
Tb=1; Ts=b*Tb; % Bit/Symbol time
Nb=16; Ns=b*Nb; % # of sample times in Tb and Ts
T=Ts/Ns; LB=4*Ns; LBN1=LB-Ns+1; % Sample time and Buffer size
ssc=[0 0; 0 1; 1 1; 1 0]; sss=ssc;
wc=8*pi/Ts; wcT=wc*T; t=[0:Ns-1]*T;
su=sqrt(2/Ts)*[cos(wc*t); -sin(wc*t)]; suT=su*T; % Basis signals
Esum= 0;
% 16-QAM signal waveforms corresponding to rectangular constellation
for i=1:L
    for l=1:L
        s(i,l,1)=2*i-L-1; s(i,l,2)=2*l-L-1; %In-phase/quadrature amplitude
        Esum= Esum +s(i,l,1)^2 +s(i,l,2)^2;
        ss(L*(l-1)+i,:)= [ssc(i,:) sss(l,:)];
        sw(L*(l-1)+i,:)=s(i,l,1)*su(1,:)+s(i,l,2)*su(2,:);
    end
end
Eav=Esum/M, Es_av=2*(M-1)/3 % Eq.(7.5.4a): Average signal energy (A=1)
Es=2; % Energy of signal waveform
A=sqrt(Es/Eav); sw=A*sw; levels=A*[-(L-1):2:L-1];
SNRdBs=[1:15]; MaxIter=10000; % Range of SNRdB and # of iterations
for iter=1:length(SNRdBs)
    SNRdB= SNRdBs(iter); SNR=10^(SNRdB/10);
    sigma2=(Es/b)/SNR; sgmsT=sqrt(sigma2/T);
    yr= zeros(2,LB); nobe= 0; % Number of bit errors to be accumulated
    for k=1:MaxIter
        im= ceil(rand*L); in= ceil(rand*L);
        imn= (in-1)*L+im; % Index of signal to transmit
        s=ss(imn,:); % Data bits to transmit
        for n=1:Ns % Operation per symbol time
            wct= wcT*(n-1); bp_noise= randn*cos(wct)-randn*sin(wct);
            rn= sw(imn,n) + sgmsT*bp_noise;
            yr= [yr(:,2:LB) suT(:,n)*rn]; % Multiplier
        end
        ycsk=sum(yr(:,LBN1:LB)'); % Sampled correlator output - DTR input
        %Detector(DTR)
        [dmin_i,mi]= min(abs(ycsk(1)-levels));
        [dmin_l,ml]= min(abs(ycsk(2)-levels));
        d= ss((ml-1)*L+mi,:); % Detected data bits
        nobe = nobe+sum(s~=d); if nobe>100; break; end
    end
    pobe(iter)= nobe/(k*b);
end
subplot(222), semilogy(SNRdbt, pobet, 'k-', SNRdBs, pobe, 'b*')
title('Probability of Bit Error for 16-ary QAM Signaling')

```

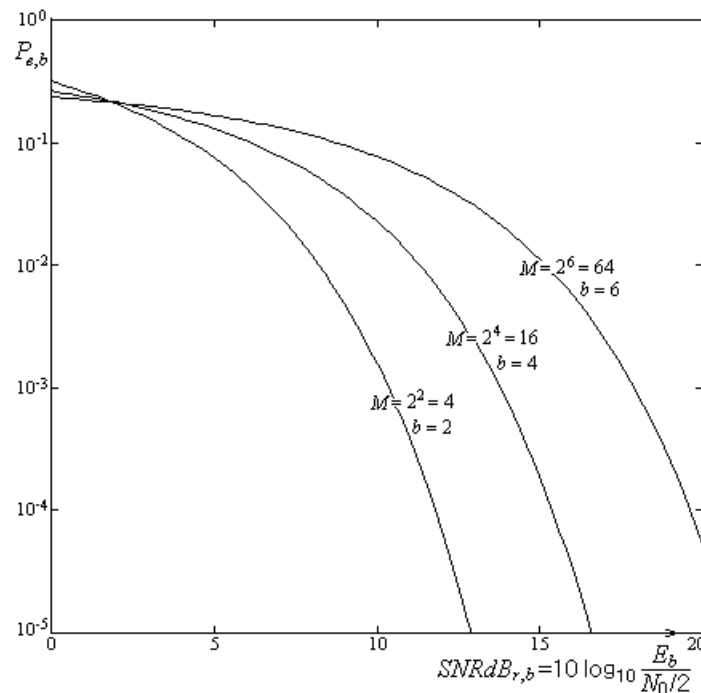
Figure 7.14 The BER curves for $M=2^b$ -ary QAM signalings

Fig. 7.14 shows the BER (bit error rate) curves, i.e. the bit error probabilities versus the average SNR per bit for $M=2^b=2^2, 2^4, 2^6$ -ary QAM signalings where the error probability tends to increase as the modulation order M increases. This tendency can be anticipated from the signal constellation diagram where the signal points get denser in the two-dimensional space and consequently, the minimum distance among the signal points gets shorter as M increases.

7.6 COMPARISON OF VARIOUS SIGNALINGS

There are several criteria to consider in deciding the signaling/modulation methods. For example,

- ◇ BER (bit error rate) performance: How low is the bit error probability for the same SNR?
- ◇ Data rate: How high is the data transmission rate[bits/sec]?
- ◇ Power efficiency: How low is the SNR required for keeping the same BER?
- ◇ Bandwidth efficiency: How narrow is the bandwidth required to keep the same data transmission rate?
- ◇ PAR (peak-to-average power ratio), interference, and out-of-band radiation
- ◇ Structural simplicity and cost: How simple is the structure and how cheap is the cost for construction and maintenance?

Table 7.1 shows the BERs for various signalings that have been discussed so far. The following MATLAB routine “`prob_error(SNRbdB,signaling,b)`” computes the error probability for SNRbdB value(s), signaling method, and number of bits per symbol. The MATLAB built-in function ‘`berawgn(EbN0dB,signaling,M)`’ is more powerful and convenient to use. The BERTool GUI (graphic user interface) can be invoked by typing ‘`bertool`’ into the MATLAB Command Window.

Table 7.1 Bit error probabilities for various signalings ($SNR_{r,b} = E_b / \sigma^2 = (E_s / b) / (N_0 / 2)$)

$M=2^b$	Signaling	Coherent (synchronous) detection	Noncoherent detection
Binary case $M=2^b$ ($b=1$)	OOK	$Q\left(\sqrt{\frac{E_s/4}{N_0/2}}\right) = Q\left(\sqrt{\frac{SNR_r}{4}}\right)$ (7.1.24)	$\frac{1}{2}e^{-SNR_r/4}$ (7.1.22)
	FSK	$Q\left(\sqrt{\frac{E_s/2}{N_0/2}}\right) = Q\left(\sqrt{\frac{SNR_r}{2}}\right)$ (7.2.9)	$\frac{1}{2}e^{-SNR_r/4}$ (7.2.20)
	PSK	$Q\left(\sqrt{\frac{E_s}{N_0/2}}\right) = Q(\sqrt{SNR_r})$ (7.3.5)	DPSK: $\frac{1}{2}e^{-SNR_r/2}$ (7.4.9)
M-ary ($b>1$)	ASK	$\frac{2(M-1)}{bM}Q\left(\sqrt{\frac{3bSNR_{r,b}}{M^2-1}}\right)$ (7.1.5/6)	-
	FSK	$\frac{M/2}{M-1}\left\{1 - \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} q(y)e^{-y^2} dy\right\}$ (7.2.8/9) with $q(y) = Q^{M-1}\left(-\sqrt{2}y - \sqrt{bSNR_{r,b}}\right)$	$\frac{M/2}{M-1} \sum_{m=0}^{M-1} (-1)^{m+1} \binom{M-1}{m} \frac{1}{m+1} e^{-\frac{mbSNR_r}{2(m+1)}}$ (7.2.19)
	PSK	$\frac{2}{b}Q\left(\sqrt{bSNR_{r,b}} \sin \frac{\pi}{M}\right)$ (7.3.8)	$\frac{2}{b}Q\left(\sqrt{bSNR_{r,b}/2} \sin \frac{\pi}{M}\right)$ (7.4.8)
	QAM	$\leq \frac{4(L-1)}{bL}Q\left(\sqrt{\frac{3b/2}{M-1}SNR_{r,b}}\right)$ (7.5.8) with $M=LN$ ($L>N$)	-

Now, let us look over the *bandwidth efficiency*, which is defined to be the ratio of the data bit (transmission) rate R_b [bits/s] over the required system bandwidth B [Hz], for $M=2^b$ -ary ASK/PSK/QAM/FSK signalings. Note the following:

- The rectangular pulse of duration D [s] in Fig. 1.1(a1) can be viewed as a signal carrying the data with symbol rate $1/D$ [symbol/s] and the null-to-null bandwidth of the rectangular pulse is $4\pi/D$ [rad/s] = $2/D$ [Hz] as can be seen from the spectrum in Fig. 1.1(a1). Thus it may be conjectured that, if a series of data with symbol rate R_s [symbol/s] is modulated with a carrier frequency ω_c and transmitted, the bandwidth is $2R_s$ [Hz].
- The interval between adjacent discrete spectra in Fig. 1.1(a1) is π/D [rad/s] = $1/2D$ [Hz], which implies that the gap between the carrier frequencies in FSK signaling is $R_s/2$ [Hz].

Therefore we can write the bandwidths and bandwidth efficiencies of ASK/PSK/QAM and (coherent) FSK signalings as

$$B_{ASK/PSK/QAM} = 2R_s = \frac{2}{b}R_b = \frac{2R_b}{b=\log_2 M}; \quad \frac{R_b}{B_{ASK/PSK/QAM}} = \frac{b=\log_2 M}{2} \quad (7.6.1)$$

$$B_{coh,FSK} = 2R_s + (M-1)\frac{R_s}{2} = \frac{(M+3)R_b}{2 \log_2 M}; \quad \frac{R_b}{B_{coh,FSK}} = \frac{2 \log_2 M}{M+3} \quad (7.6.2)$$

```

function p=prob_error(SNRbdB,signaling,b,opt1,opt2)
% Finds the symbol/bit error probability for given SNRbdB (Table 7.1)
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
if nargin<5, opt2='coherent'; end % opt2='coherent' or 'noncoherent'
if nargin<4, opt1='SER'; end % opt1='SER' or 'BER'
M=2^b; SNRb=10.^(SNRbdB/10); NSNR=length(SNRb);
if signaling(1:3)=='ASK' % ASK (PAM)
    if lower(opt2(1))=='c' % ASK coherent --> Eq.(7.1.5)
        for i=1:NSNR, p(i)=2*(M-1)/M*Q(sqrt(3*b*SNRb(i)/(M^2-1))); end
        if lower(opt1(1))=='b', p = p/b; end
    else % ASK noncoherent --> Eq.(7.1.22)
        if b==1, for i=1:NSNR, p(i)=exp(-SNRb(i)/4)/2; end; end
    end
elseif signaling(1:3)=='FSK'
    tmp=M/2/(M-1);
    f5251_ = inline('Q(-sqrt(2)*x-sqrt(b*SNRb)).^(2^b-1)', 'x', 'SNRb', 'b');
    if lower(opt2(1))=='c' % FSK coherent
        if b==1
            for i=1:NSNR, p(i)=Q(sqrt(SNRb(i)/2)); end %Eq.(7.2.9)
        else
            for i=1:NSNR
                p(i) = 1-Gauss_Hermite(f5251_,10,SNRb(i),b)/sqrt(pi);
            end
        end
    else % FSK noncoherent
        for i=1:NSNR
            p(i)=(M-1)/2*exp(-b*SNRb(i)/4); tmp1=M-1;
            for m=2:M-1
                tmp1=-tmp1*(M-m)/m;
                p(i)=p(i)+tmp1/(m+1)*exp(-m*b*SNRb(i)/2/(m+1)); % Eq.(7.2.19)
            end
        end
    end
    if lower(opt1(1))=='b'&b>1, p = p*tmp; end
elseif signaling(1:3)=='PSK' % Eq.(7.3.7)
    for i=1:NSNR, p(i)=(1+(b>1))*Q(sqrt(b*SNRb(i))*sin(pi/M)); end
    if lower(opt1(1))=='b'&b>1, p = p/b; end
elseif signaling(1:3)=='DPS' % DPSK
    if b==1 % Eq.(7.4.8)
        for i=1:NSNR, p(i)=2*Q(sqrt(b*SNRb(i)/2)*sin(pi/M)); end
    else
        for i=1:NSNR, p(i)=exp(-SNRb(i)/2)/2; end %Eq.(7.4.9)
        if lower(opt1(1))=='b', p = p/b; end
    end
elseif signaling(1:3)=='QAM'
    L=2^(ceil(b/2)); N = M/L;
    for i=1:NSNR
        tmpL = 1-2*(L-1)/L*Q(sqrt(3*b/2/(L^2-1)*SNRb(i)));
        tmpN = 1-2*(N-1)/N*Q(sqrt(3*b/2/(N^2-1)*SNRb(i)));
        p(i) = 1-tmpL*tmpN; % Eq.(7.5.6)
    end
    if lower(opt1(1))=='b'&b>1, p = p/b; end
end

```

Table 7.2 Power efficiency and bandwidth efficiency with signaling

The number of bits per symbol	<Power efficiency> $SNRdB_{r,b} = 10 \log_{10}(E_b/(N_0/2))$ [dB] for $P_{e,b} = 10^{-5}$				<Bandwidth efficiency> R_b / B [bits/s/Hz]		
	ASK	PSK	FSK	QAM	ASK/PSK	FSK	QAM
$b=1$	12.6	12.6	15.6	-	0.5000	0.4000	-
$b=2$	16.8	12.9	13.1	12.9	1.0000	0.5714	1.0000
$b=3$	21.3	16.5	11.6	-	1.5000	0.5455	-
$b=4$	26.1	21.1	10.7	17.1	2.0000	0.4211	2.0000
$b=5$	31.2	26.1	9.9	-	2.5000	0.2857	-
$b=6$	36.5	31.3	9.4	21.6	3.0000	0.1791	3.0000
$b=7$	41.8	36.7	8.9	-	3.5000	0.1069	-
$b=8$	47.3	42.1	8.5	26.4	4.0000	0.0618	4.0000

```

%dc07t02.m
% fills in Table 7.2 with SNRdB for BER=1e-5 and bandwidth efficiency
clear, clf
tol=1e-14; bs=[1:8];
% Define a nonlinear equation to be solved for SNRdB using fsolve()
nonlinear_eq=inline('prob_error(x,signaling,b)-1e-5','x','signaling','b');
% For PSK signaling
for i=1:length(bs)
    b=bs(i); RB_PSK(i)=b/2; % Bandwidth efficiency
    if i==1, x0=10; else x0=SNRbdBs_PSK(i-1); end
    SNRbdBs_PSK(i)=fsolve(nonlinear_eq,x0,optimset('TolFun',tol),'PSK',b);
end
disp('PSK'), [bs; SNRbdBs_PSK; RB_PSK]
% For FSK signaling
for i=1:length(bs)
    b=bs(i); RB_FSK(i)=2*b/(2^b+3);
    if i==1, x0=10; else x0=SNRbdBs_FSK(i-1); end
    SNRbdBs_FSK(i)=fsolve(nonlinear_eq,x0,optimset('TolFun',tol),'FSK',b);
end
disp('FSK'), [bs; SNRbdBs_FSK; RB_FSK]
% For QAM signaling
bs_QAM=[2:2:8];
for i=1:length(bs_QAM)
    b=bs_QAM(i); RB_QAM(i)=b/2;
    if i==1, x0=10; else x0=SNRbdBs_QAM(i-1); end
    SNRbdBs_QAM(i)=fsolve(nonlinear_eq,x0,optimset('TolFun',tol),'QAM',b);
end
disp('QAM'), [bs_QAM; SNRbdBs_QAM; RB_QAM]
% For ASK signaling
for i=1:length(bs)
    b=bs(i); RB_ASK(i)=b/2;
    if i==1, x0=10; else x0=SNRbdBs_ASK(i-1); end
    SNRbdBs_ASK(i)=fsolve(nonlinear_eq,x0,optimset('TolFun',tol),'ASK',b);
end
disp('ASK'), [bs; SNRbdBs_ASK; RB_ASK]

```

Table 7.2 shows the *power efficiency*, i.e. the SNR[dB] required to achieve the BER of 10^{-5} and the *bandwidth efficiency*, i.e. the ratio of data transmission rate[bits/s] over bandwidth[Hz], that are also depicted in Fig. 9.7. As the modulation order M or the number b of bits per symbol increases, FSK tends to have higher power efficiency and lower bandwidth efficiency while PSK tends to have lower power efficiency and higher bandwidth efficiency, as can also be seen from Figs. 7.5 and 7.10. The above MATLAB program “dc07t02.m” can be used to compute the values of $SNRdB_{r,b}$ at which PSK/FSK/QAM/ASK signalings achieve the BER of 10^{-5} .

Before ending this section, it would be well worth to see the usage of the BER computing function ‘berawgn()’ in the MATLAB Help manual:

```
ber = berawgn(EbNo,'PAM',M)
Ber = berawgn(EbNo,'QAM',M)
Ber = berawgn(EbNo,'PSK',M,dataenc) with dataenc='diff'/'nondiff' for differential/non-differential
ber = berawgn(EbNo,'OQPSK',dataenc) for OQPSK (offset QPSK) (see Problem 7.4)
ber = berawgn(EbNo,'DPSK',M)
ber = berawgn(EbNo,'FSK',M,coh) with coh='coherent'/'noncoherent' for coherent/noncoherent
ber = berawgn(EbNo,'FSK',2,coh,rho) with rho=the complex correlation coefficient
ber = berawgn(EbNo,'MSK',dataenc) with dataenc='diff'/'nondiff' for differential/non-differential
ber = berawgn(EbNo,'MSK',dataenc,coherence)
berlb = berawgn(EbNo,'CPFSK',M,modindex,kmin) % gives the BER lowerbound
```

Problems

7.1 Linear Combination of Gaussian Noises

Suppose we have a zero-mean white Gaussian noise $n(t)$ of mean and variance as

$$E\{n(t)\} = 0 \quad (\text{P7.1.1})$$

$$\text{cov}\{n(\tau), n(t)\} = E\{n(\tau)n(t)\} = \sigma^2 \delta(\tau - t) \quad (\text{P7.1.2})$$

Also, consider another noise

$$n_1(t) = \int_0^T n(\tau + t) s_u(\tau) d\tau \quad (\text{P7.1.3})$$

which is obtained from taking the crosscorrelation of a zero-mean white noise $n(t)$ with a unit energy signal $s_u(t)$ such that

$$\int_0^T s_u^2(\tau) d\tau = 1 \quad (\text{P7.1.4})$$

It is claimed that this is also a zero-mean Gaussian noise with variance σ^2 .

- (a) To verify this, check the following derivation of the mean and covariance of $n_1(t)$. Note that a linear combination of Gaussian processes is also Gaussian.

```
%dc07p01.m
% See if a linear convolution of Gaussian noises
% with a unit signal waveform is another Gaussian noise
clear, clf
K=10000; % # of iterations for getting the error probability
Ts=1; N=40; T=Ts/N; % Symbol time and Sample time
N4=N*4; % Buffer size of correlator
wc=10*pi/Ts; t=[0:N-1]*T; wct=wc*t;
su=sqrt(2/Ts)*cos(wct); % Unit energy signal
signal_power=su*sus'*T % Signal energy
sigma2=2; sigma=sqrt(sigma2); sqT=sqrt(T);
noise= zeros(1,N4); % Noise buffer
for k=1:K
    for n=1:N % Operation per symbol time
        noise0= sigma*randn;
        noise=[noise(2:N4) noise0/sqT]; % Bandpass noise
        noise1= su*noise(3*N+1:N4) '*T;
    end
    noise0s(k)=noise0; noise1s(k)=noise1;
end
phi0=xcorr1(noise0s); phi1=xcorr1(noise1s);
plot(phi0), hold on, pause, plot(phi1, 'r')
```

(proof)

$$E\{n_1(t)\} = E\left\{\int_0^{T_s} n(\tau+t)s_u(\tau) d\tau\right\} = \int_0^{T_s} E\{n(\tau+t)\}s_u(\tau) d\tau \stackrel{(P7.1.1)}{=} 0 \quad (P7.1.5)$$

$$\begin{aligned} E\{n_1^2(t)\} &= E\left\{\int_0^{T_s} n(\tau+t)s_u(\tau) d\tau \int_0^{T_s} n(v+t)s_u(v) dv\right\} \\ &= \int_0^{T_s} \int_0^{T_s} s_u(\tau)s_u(v) E\{n(\tau+t)n(v+t)\} d\tau dv \\ &\stackrel{(P7.1.2)}{=} \sigma^2 \int_0^{T_s} \int_0^{T_s} s_u(\tau)s_u(v) \delta(\tau-v) d\tau dv \stackrel{(E1.6.1)}{=} \sigma^2 \int_0^{T_s} s_u^2(\tau) d\tau \stackrel{(P7.1.4)}{=} \sigma^2 \quad (P7.1.6) \end{aligned}$$

- (b) As an alternative for verification, the above program “dc07p01.m” is composed to generate a noise $n(t)$ with Gaussian distribution $N(m=0, \sigma^2=2)$ and take the sampled crosscorrelation between $n(t)$ and a unit energy signal $s_u(t)$ to make another noise $n_1(t)$. Plot the autocorrelation of these two noises (noise0 and noise1) and make a comment on their closeness or similarity of their statistical properties such as the mean and variance.

7.2 Coherent/Noncoherent Detection with Time Difference between XMTR and RCVR

- (a) In order to feel how seriously the time difference between XMTR and RCVR affects the BER performance of a communication system, use the following statements to modify the MATLAB program “sim_FSK_passband_coherent.m” so that it can accommodate some delay time or time difference (td) between XMTR and RCVR clocks. Then run the modified program to see the changed BER curve. Does it stay away from the theoretical BER curve?

```
nd=2; % Number of delay samples
sws=zeros(1, LB);
sws=[sws(2:LB) sw(i,n)];
r=[r(2:LB) sws(end-nd)+sgmst*bp_noise]; % Received signal
```

- (b) In order to see that noncoherent detection is of relative advantage over coherent detection in facing the time difference between XMTR and RCVR, replace $nd=0$ by $nd=2$ in the 14th line of the program “sim_FSK_passband_noncoherent.m” (Sec. 7.2) and run the modified program to see the changed BER curve. Does it still touch the theoretical BER curve?
- (c) Modify the 19th line of the QPSK simulation program “sim_PSK_passband.m” (Sec. 7.3) into $nd=1$ and run the program to see the changed constellation diagram and BER curve. Modify the 16th line of the QDPSK simulation program “sim_DPSK_passband.m” (Sec. 7.4) into $nd=1$ and run the program to see the BER curve. What is implied by the simulation results?

7.3 $M=2^2$ -ary QAM (Quadrature AM) and $\pi/4$ -QPSK(Quadrature PSK)

From the QAM signal constellation diagrams in Fig. 7.11, it can be seen that $M=2^2$ -ary QAM is the same as $\pi/4$ -QPSK, whose signal constellation diagram is a rotation of the QPSK signal constellation diagram (Fig. 7.1(b3)) by $\pi/4=45^\circ$ as depicted in Fig. P7.5(a2). Is it also supported by the conformity of the error probabilities (between the standard QPSK and the $\pi/4$ -QPSK) that turned out to be Eq. (7.3.7) and Eq. (7.5.7), respectively? You can substitute $b=2$, $L=2^{b/2}=2$, and $M=2^2$ into both of the equations and check if they conform with each other.

7.4 OQPSK (Offset Quadrature PSK) or SQPSK (Staggered QPSK)

As depicted in Fig. P7.4, OQPSK is a slight modification of QPSK (Fig. 7.12) in such a way that the Q(uadrature)-channel bit stream is offset w.r.t. the I(n-phase)-channel by a bit duration (Fig. P7.4(b2)). Compared with QPSK, it allows no simultaneous change of two bits to prevent any state transition accompanying the phase change of 180° (Fig. P7.4(c2)) (in the teeth of making more frequent state transitions possibly every bit time) and consequently, the envelope variation becomes less severe so that the transmitted signal bandwidth can be limited more strictly.

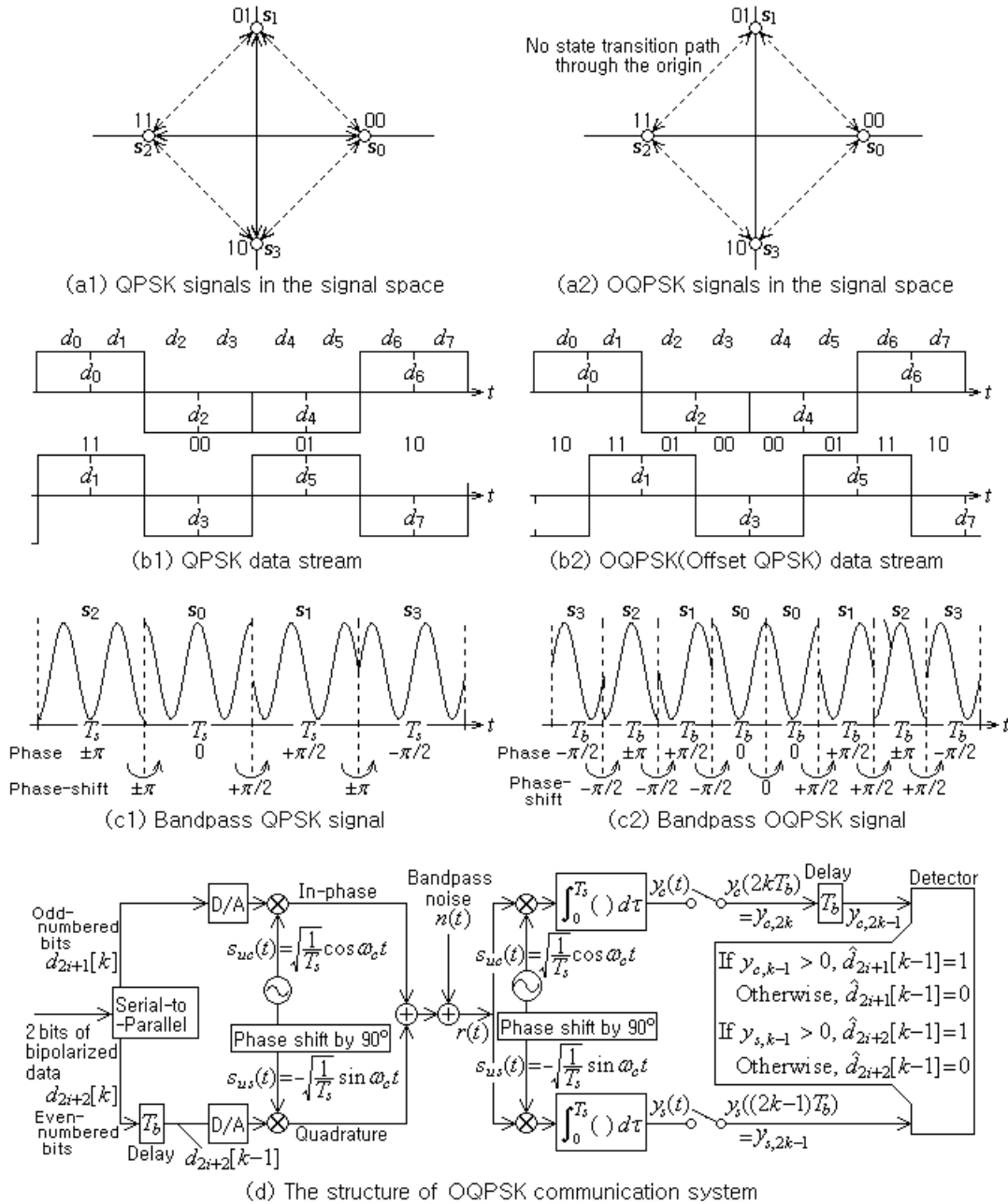


Figure P7.4 QPSK and OQPSK communication systems

```

%sim_OQPSK.m
% simulates a digital communication system
% with O(ffset)QPSK signal waveforms in Fig.P7.4
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
clear, clf
b=2; M=2^b;
SNRdbBt=0:0.1:10; SNRbt=10.^(SNRdbBt/10);
pobet=prob_error(SNRdbBt, 'PSK', b, 'bit');
Tb=1; Ts=b*Tb; % Bit/symbol time
Nb=16; Ns=Nb*b; % # of sample times in Tb and Ts
T=Ts/Ns; LB=4*Ns; LBN1=LB-Ns+1; % Sample time and Buffer size
Es=2; sqEb=sqrt(Es/b); % Energy of signal waveform
% QPSK signal waveforms
ss=[0 0; 0 1; 1 1; 1 0];
wc=2*pi/Ts; wcT=wc*T; t=[0:Ns-1]*T;
su=sqrt(2/Ts)*[cos(wc*t); ??????????]; suT=su*T;
sw=sqEb*su;
SNRdBs=[1:3:10]; MaxIter=10000; % Range of SNRdB, # of iterations
for iter=1:length(SNRdBs)
    SNRdB=SNRdBs(iter); SNRb=10^(SNRdB/10);
    N0=2*(Es/b)/SNRb; sigma2=N0/2; sgmsT=sqrt(sigma2/T);
    yr= zeros(2, LB); yc=zeros(1,2); ys=zeros(1,2);
    iq=0; % Initialize the quadrature bit arbitrarily
    nobe=0; % Number of bit errors to be accumulated
    for k=1:MaxIter
        i=ceil(rand*M); s=ss(i,:); wct=-wcT;
        for n=1:b % Operation per symbol time
            if n==1, ii=2*ss(i,1)-1; % In-phase bit
                else iq=2*ss(i,2)-1; % Quadrature bit
            end
            mn=0;
            for m=1:Nb % Operation per bit time
                wct= wct+wcT; mn=mn+1;
                bp_noise= randn*cos(wct)-randn*sin(wct);
                rn=ii*sw(1,mn)+iq*sw(2,mn)+sgmsT*bp_noise;
                yr=[yr(:,2:LB) suT(:,mn)*rn]; % Multiplier
            end
            if n==2 % sampled at t=2k*Tb
                yc=[yc(2) sum(yr(:,LBN1:LB))]; % Correlator output
            else % sampled at t=(2k-1)*Tb
                ys=[ys(2) sum(yr(:,LBN1:LB))]; % Correlator output
            end
        end
        d=( [yc(?) ys(?)] > 0); % Detector(DTR)
        if k>1, nobe=nobe+sum(s0~=d); end
        if nobe>100, break; end
        s0= s;
    end
    pobe(iter)= nobe/(k*b);
end
pobe
semilogy(SNRdbBt, pobet, 'k-', SNRdBs, pobe, 'b*')
title('Probability of Bit Error for (4-ary) QPSK Signaling')

```


The above MATLAB program “sim_OQPSK.m” is a modification of the program “sim_PSK_passband.m” (QPSK) (in Sec. 7.3) to simulate the OQPSK communication system, whose block diagram is depicted in Fig. P7.4, but it is unfinished. Finish it up by replacing the three parts of ?’s with appropriate statements or just indices and run it to see the BER curve.

7.5 $\pi/4$ -Shifted QPSK (Quadrature PSK)

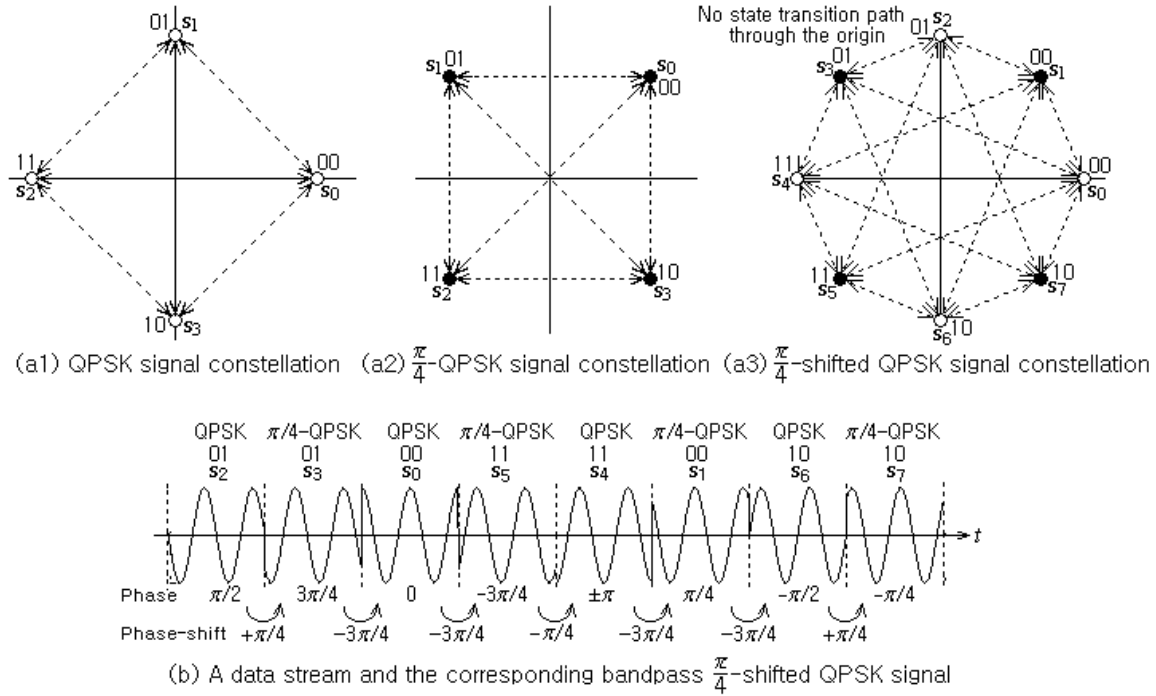


Figure P7.5 $\pi/4$ -shifted QPSK (Quadrature Phase Shift Keying) signaling

$\pi/4$ -shifted QPSK is another way of lowering envelope fluctuations than OQPSK discussed in Problem 7.4. It assigns one of the signal points in the (non-offset) QPSK (Fig. P7.5(a1)) and $\pi/4$ -QPSK (Fig. P7.5(a2)) signal constellations to even and odd-number indexed data symbols as depicted in Fig. P7.5(b) and therefore, it virtually uses the dual signal constellation diagram shown in Fig. P7.5(a3) where the modulation causes no state transition path through the origin as in OQPSK. Compared with OQPSK, it has the maximum phase change of $\pm 3\pi/4$, which is larger than that ($\pm\pi/2$) of OQPSK; On the other hand, differential encoding/decoding with noncoherent detection can be implemented in $\pi/4$ -shifted QPSK signaling, since it has four possible phase-shifts of $\Delta\theta = \pm\pi/4$ and $\pm 3\pi/4$ (as can be seen from Fig. P7.5(b)) while OQPSK has only two possible phase-shifts of $\Delta\theta = \pm\pi/2$.

Table P7.5 Message data dibits and the corresponding phase-shifts in $\pi/4$ -shifted QPSK

(Gray-coded) message dibits	Phase-shifts	$s_k = s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7$
$d_{2k-1}d_{2k} = 00$	$\Delta\theta = +\pi/4$	$s_{(k+1)\bmod 8} = s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_0$
$d_{2k-1}d_{2k} = 01$	$\Delta\theta = +3\pi/4$	$s_{(k+3)\bmod 8} = s_3 s_4 s_5 s_6 s_7 s_0 s_1 s_2$
$d_{2k-1}d_{2k} = 11$	$\Delta\theta = -3\pi/4$	$s_{(k+5)\bmod 8} = s_5 s_6 s_7 s_0 s_1 s_2 s_3 s_4$
$d_{2k-1}d_{2k} = 10$	$\Delta\theta = -\pi/4$	$s_{(k+7)\bmod 8} = s_7 s_0 s_1 s_2 s_3 s_4 s_5 s_6$

- (a) Modify the (standard) QPSK simulation program “sim_PSK_passband.m” into a $\pi/4$ -shifted QPSK simulation program named, say, “sim_S_QPSK.m” and run it to see the BER curve. Does it conform with the theoretical BER curve?
- (b) Referring to Table P7.5, modify the (standard) QDPSK (quadrature differential phase shift keying) simulation program “sim_DPSK_passband.m” into a $\pi/4$ -shifted QDPSK simulation program named, say, “sim_S_QDPSK.m” and run it to see the BER curve. Does it conform with the theoretical one? If you have no idea, start with the following unfinished program:

```
%sim_S_QDPSK.m
% simulates the pi/4-shifted QDPSK signaling (Table P7.5)
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
b=2; M=2^b; M2=M*2;
SNRdbT=0:0.1:10; SNRbt=10.^(SNRdbT/10);
pobet=prob_error(SNRdbT, 'DPSK', b, 'bit');
Tb=1; Ts=b*Tb; % Bit/symbol time
Nb=16; Ns=b*Nb; % Numbers of sample times in Tb and Ts
T=Ts/Ns; LB=4*Ns; LBN1=LB-Ns+1; % Sample time and Buffer size
Es=2; % Energy of signal waveform
% QDPSK signal waveforms
ss=[0 0; 0 1; 1 1; 1 0];
wc=8*pi/Ts; wct=wc*T; t=[0:Ns-1]*T; nd=1;
for m=1:M2, sw(m,:)=sqrt(2*Es/Ts)*cos(wc*t+(m-1)*pi/M); end
su= sqrt(2/Ts)*[cos(wc*t); -sin(wc*t)]; suT=su*T;
SNRdBs=[1:3:10]; MaxIter=10000; %Range of SNRdB, # of iterations
for iter=1:length(SNRdBs)
    SNRdB=SNRdBs(iter); SNRb=10^(SNRdB/10);
    N0=2*(Es/b)/SNRb; sigma2=N0/2; sgmsT=sqrt(sigma2/T);
    sws=zeros(1, LB); yr=zeros(2, LB);
    nobe=0; % Number of bit errors to be accumulated
    is0=1; % Initial signal index
    th0=1; % Initial guess (possibly wrong)
    for k=1:MaxIter
        i= ceil(rand*M); s=ss(i,:); % Data bits to transmit
        is= mod(is0+????, M2)+1; % Signal to transmit (Table P7.5)
        for n=1:Ns % Operation per symbol time
            sws=[sws(2:LB) sw(is,n)];
            wct=wct*(n-1); bp_noise= randn*cos(wct)-randn*sin(wct);
            rn= sws(end-nd) + sgmsT*bp_noise;
            yr=[yr(:, 2:LB) suT(:, n)*rn]; % Multiplier
        end
        ycsk=sum(yr(:, LBN1:LB)')'; % Sampled correlator output
        %Detector(DTR)
        th=atan2(ycsk(2), ycsk(1)); dth=th-th0;
        if dth<0, dth=dth+2*pi; end
        [themin, lmin]=min(abs(dth- [????]*2*pi/M));
        d= ss(lmin,:); % Detected data bits
        nobe= nobe+sum(s~=d); if nobe>100, break; end
        is0=is; th0=th; % update the previous signal and theta
    end
    pobe(iter)= nobe/(k*b);
end
semilogy(SNRdbT, pobet, 'k', SNRdBs, pobe, 'b')
```

7.6 Minimum-Shift Keying (MSK)

Minimum-shift keying is a type of continuous phase frequency shift keying (CP-FSK). It encodes each data bit into a half sinusoid

$$s(t) = \sqrt{\frac{2E_b}{T_b}} \cos(\omega_c t + \theta(t)) \quad (\text{P7.6.1})$$

with its phase changed continuously as

$$\theta(t) = \theta(kT_b) \pm \frac{\pi}{2T_b} (t - kT_b) \quad \text{with } \theta(0) = 0 \quad (\text{P7.6.2})$$

where the second term has a plus or minus sign for each bit interval depending on whether the value of message data bit is 1 or 0 (see Fig. P7.6). As a consequence, the phase of the signal waveform at the boundaries between two consecutive bit intervals becomes

$$\theta(2kT_b) = \pm m\pi \quad \text{or} \quad \theta((2k+1)T_b) = \pm \frac{\pi}{2} \pm m\pi \quad (\text{P7.6.3})$$

and the instantaneous frequency, which is obtained by differentiating the angular argument of $s(t)$ w.r.t. t , becomes

$$\omega_1 = \omega_c + \frac{\pi}{2T_b} \quad \text{or} \quad \omega_0 = \omega_c - \frac{\pi}{2T_b} \quad (\text{P7.6.4})$$

Note that the difference of these two frequencies is π/T_b and it is the same as the minimum frequency gap required to keep the orthogonality between two frequency components for a bit duration T_b (see the discussion just below Eq. (7.2.4)).

To find two (orthogonal) basis signal waveforms that can be used to detect the phase of the received signal, we rewrite Eq. (P7.6.1) as

$$s(t) \stackrel{(\text{P7.6.1})}{=} \sqrt{\frac{2E_b}{T_b}} (\cos \theta(t) \cos \omega_c t - \sin \theta(t) \sin \omega_c t) = s_I(t) \cos \omega_c t - s_Q(t) \sin \omega_c t \quad (\text{P7.6.5})$$

where

$$\begin{aligned} s_I(t) &= \sqrt{\frac{2E_b}{T_b}} \cos \theta(t) \quad \text{with } \theta(t) \stackrel{(\text{P7.6.2})}{=} \theta(2kT_b) \pm \frac{\pi}{2T_b} (t - 2kT_b) \\ &\stackrel{(\text{D.20})}{=} \sqrt{\frac{2E_b}{T_b}} \left(\cos \theta(2kT_b) \cos \left(\pm \frac{\pi}{2T_b} (t - 2kT_b) \right) - \sin \theta(2kT_b) \sin \left(\pm \frac{\pi}{2T_b} (t - 2kT_b) \right) \right) \\ &= \sqrt{\frac{2E_b}{T_b}} \cos \theta(2kT_b) \cos \left(\pm \frac{\pi}{2T_b} t \right) \quad \text{for } (2k-1)T_b < t \leq (2k+1)T_b \\ &= \begin{cases} +\sqrt{\frac{2E_b}{T_b}} \cos \left(\frac{\pi}{2T_b} t \right) & \text{if } \theta(2kT_b) = 0 \\ -\sqrt{\frac{2E_b}{T_b}} \cos \left(\frac{\pi}{2T_b} t \right) & \text{if } \theta(2kT_b) = \pm\pi \end{cases} \end{aligned} \quad (\text{P7.6.6a})$$

$$\begin{aligned}
s_Q(t) &= \sqrt{\frac{2E_b}{T_b}} \sin \theta(t) \quad \text{with } \theta(t) \stackrel{(P7.6.2)}{=} \theta((2k+1)T_b) \pm \frac{\pi}{2T_b}(t - (2k+1)T_b) \\
&\stackrel{(D.19)}{=} \sqrt{\frac{2E_b}{T_b}} \begin{pmatrix} \sin \theta((2k+1)T_b) \cos \left(\pm \frac{\pi}{2T_b}(t - (2k+1)T_b) \right) \\ \pm \cos \theta((2k+1)T_b) \sin \left(\pm \frac{\pi}{2T_b}(t - (2k+1)T_b) \right) \end{pmatrix} \\
&= \sqrt{\frac{2E_b}{T_b}} \sin \theta((2k+1)T_b) \sin \left(\pm \frac{\pi}{2T_b}t \right) \quad \text{for } 2kT_b < t \leq 2(k+1)T_b \\
&= \begin{cases} +\sqrt{\frac{2E_b}{T_b}} \sin \left(\frac{\pi}{2T_b}t \right) & \text{if } \theta((2k+1)T_b) = +\frac{\pi}{2} \\ -\sqrt{\frac{2E_b}{T_b}} \sin \left(\frac{\pi}{2T_b}t \right) & \text{if } \theta((2k+1)T_b) = -\frac{\pi}{2} \end{cases} \tag{P7.6.6b}
\end{aligned}$$

Substituting these two equations (P7.6.6a) and (P7.6.6b) into Eq. (P7.6.5) yields

$$s(t) = \pm \sqrt{\frac{2E_b}{T_b}} \cos \left(\frac{\pi}{2T_b}t \right) \cos \omega_c t \mp \sqrt{\frac{2E_b}{T_b}} \sin \left(\frac{\pi}{2T_b}t \right) \sin \omega_c t \tag{P7.6.7}$$

This implies that the basis signal waveforms to be used for detection at RCVR are

$$s_{uc}(t) = \sqrt{\frac{2}{T_b}} \cos \left(\frac{\pi}{2T_b}t \right) \cos \omega_c t \tag{P7.6.8a}$$

$$s_{us}(t) = \sqrt{\frac{2}{T_b}} \sin \left(\frac{\pi}{2T_b}t \right) \sin \omega_c t \tag{P7.6.8b}$$

Therefore, RCVR computes the correlation between the received signal $r(t)$ and these two basis signal waveforms and samples it at $t = 2kT_b$ and $(2k+1)T_b$ to get

$$\begin{aligned}
y_c((2k+1)T_b) &\stackrel{(5.1.27)}{=} \int_{-T_b}^{T_b} s_{uc}(t) r(t + 2kT_b) dt = \int_{-T_b}^{T_b} s_{uc}(t) s(t + 2kT_b) dt + n_c \\
&\stackrel{(P7.6.1)}{=} \int_{-T_b}^{T_b} \sqrt{\frac{2}{T_b}} \cos \left(\frac{\pi}{2T_b}t \right) \cos \omega_c t \cos(\omega_c t + \theta(2kT_b)) dt + n_c \\
&\stackrel{(P7.6.8a)}{=} \int_{-T_b}^{T_b} \sqrt{\frac{2}{T_b}} \cos \left(\frac{\pi}{2T_b}t \right) \cos(\omega_c t + \theta(2kT_b)) dt + n_c \\
&\stackrel{(D.25)}{=} \frac{\sqrt{E_b}}{T_b} \int_{-T_b}^{T_b} \cos \left(\frac{\pi}{2T_b}t \right) (\cos(2\omega_c t + \theta(2kT_b)) + \cos \theta(2kT_b)) dt + n_c \tag{P7.6.9a} \\
&\stackrel{(D.25)}{=} \frac{\sqrt{E_b}}{2T_b} \int_{-T_b}^{T_b} \begin{pmatrix} \cos \left(2\omega_c t + \theta(2kT_b) + \frac{\pi}{2T_b}t \right) + \cos \left(2\omega_c t + \theta(2kT_b) - \frac{\pi}{2T_b}t \right) \\ + \cos \left(\theta(2kT_b) + \frac{\pi}{2T_b}t \right) + \cos \left(\theta(2kT_b) - \frac{\pi}{2T_b}t \right) \end{pmatrix} dt + n_c \\
&= \sqrt{E_b} \cos(\theta(2kT_b)) + n_c
\end{aligned}$$

$$\begin{aligned}
y_s(2kT_b) &\stackrel{(5.1.27)}{=} \int_0^{2T_b} s_{us}(t) r(t + (2k-1)T_b) dt = \int_0^{2T_b} s_{us}(t) s(t + (2k-1)T_b) dt + n_s \\
&\stackrel{(P7.6.1)}{=} \sqrt{\frac{2E_b}{T_b}} \sqrt{\frac{2}{T_b}} \int_0^{2T_b} \sin\left(\frac{\pi}{2T_b}t\right) \sin \omega_c t \cos(\omega_c t + \theta((2k-1)T_b)) dt + n_s \\
&\stackrel{(P7.6.8b)}{=} \sqrt{\frac{E_b}{T_b}} \int_0^{2T_b} \sin\left(\frac{\pi}{2T_b}t\right) (\sin(2\omega_c t + \theta((2k-1)T_b)) - \sin \theta((2k-1)T_b)) dt + n_s \quad (P7.6.9b) \\
&\stackrel{(D.22)}{=} \frac{\sqrt{E_b}}{2T_b} \int_0^{2T_b} \left(\cos\left(2\omega_c t + \theta((2k-1)T_b) - \frac{\pi}{2T_b}t\right) - \cos\left(2\omega_c t + \theta((2k-1)T_b) + \frac{\pi}{2T_b}t\right) \right. \\
&\quad \left. + \cos\left(\theta((2k-1)T_b) + \frac{\pi}{2T_b}t\right) - \cos\left(\theta((2k-1)T_b) - \frac{\pi}{2T_b}t\right) \right) dt + n_s \\
&= -\sqrt{E_b} \sin(\theta((2k-1)T_b)) + n_s
\end{aligned}$$

Depending on the sign of these sampled values of correlator outputs, the phase estimates are determined as

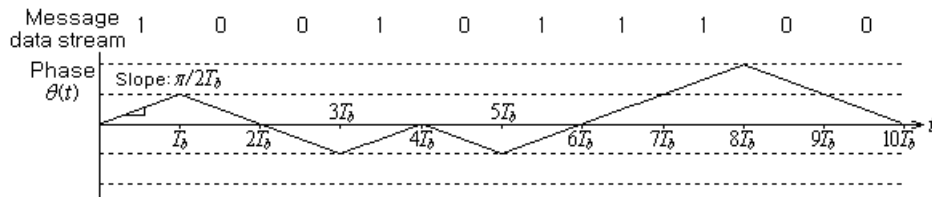
$$\theta(2kT_b) = \begin{cases} 0 & \text{if } y_c((2k+1)T_b) > 0 \\ \pm\pi \text{ (close to } \theta((2k-1)T_b)) & \text{if } y_c((2k+1)T_b) < 0 \end{cases} \quad (P7.6.10a)$$

$$\theta((2k-1)T_b) = \begin{cases} -\pi/2 & \text{if } y_s(2kT_b) > 0 \\ +\pi/2 & \text{if } y_s(2kT_b) < 0 \end{cases} \quad (P7.6.10b)$$

so that the maximum difference between the phase estimates at successive sampling instants is $\pm\pi/2$. Then the detector judges the value of transmitted data bit to be 1 or 0 depending on whether θ turns out to have increased or decreased.

Noting that the sampled values of correlator outputs $y_c((2k+1)T_b)$ and $y_s(2kT_b)$ will be distributed around $(+\sqrt{E_b}, 0)$ and $(-\sqrt{E_b}, 0)$ in the signal space and thus the (minimum) distance between signal points is $d_{\min} = 2\sqrt{E_b}$, the (symbol or bit) error probability is

$$P_{e,b} \stackrel{(5.2.35)}{=} Q\left(\frac{d_{\min}/2}{\sigma = \sqrt{N_0/2}}\right) = Q\left(\sqrt{\frac{E_b}{N_0/2}}\right) = Q(\sqrt{SNR_{r,b}}) \quad (P7.6.11)$$



(a) (Continuous) change of the phase of an MSK signal

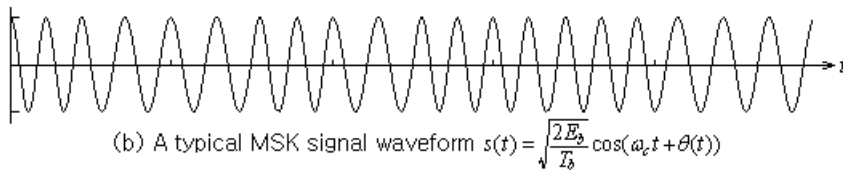


Figure P7.6 The phase and waveform of an MSK signal

```

%sim_MSK.m
% simulates a digital communication system with MSK signaling
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
b=1; M=2^b; % # of bits per symbol and modulation order
SNRbDbt=0:0.1:12; SNRbt=10.^(SNRbDbt/10);
pbe_PSK= prob_error(SNRbDbt, 'PSK', b, 'bit');
pbe_FSK= prob_error(SNRbDbt, 'FSK', b, 'bit');
Tb=1; Ts=b*Tb; % Bit/Symbol time
Nb=16; Ns=Nb*b; % # of sample times in Tb and Ts
T=Ts/Ns; LB=4*Ns; LBN2=LB-2*Ns+1; % Sample time and Buffer size
Es=2; % Energy of signal waveform
% QPSK signal waveforms
ds=[0 1];
wc=2*pi; wct=wc*T;
t=[0:2*Nb-1]*T; pihTbt=pi/2/Tb*t;
su=sqrt(2/Tb)*[cos(pihTbt-pi/2).*cos(wc*(t-Tb))];
% ??????????????????????]; % Eq. (P7.6.8)
su=[fftshift(su(1,:)); su(2,:)]; suT=su*T;
ik=[1 2 1 2 1 1 2 2 2 1];
sq2EbTb=sqrt(2*Es/b/Tb); pihTbT=pi/2/Tb*T;
SNRdBs=[1 3 10]; % Range of SNRdB
MaxIter=10000; % Number of iterations
for iter=1:length(SNRdBs)
    SNRdB=SNRdBs(iter); SNR=10^(SNRdB/10);
    N0=2*(Es/b)/SNR; sigma2=N0/2; sgmsT=sqrt(sigma2/T);
    yr=zeros(2, LB); yc=0; ys=0; th0=0; t=0; wct=0; tht=th0; mn=1;
    nobe= 0; % Number of bit errors to be accumulated
    for k=1:MaxIter
        i=ceil(rand*M); s(k)=ds(i); sgn=s(k)*2-1;
        for m=1:Nb % Operation per bit time
            bp_noise= randn*cos(wct)-randn*sin(wct);
            rn= sq2EbTb*cos(wct+tht) + sgmsT*bp_noise;
            yr=[yr(:,2:LB) suT(:,mn)*rn]; % Correlator output - DTR input
            t=t+T; wct=wct+wct; tht=tht+sgn*pihTbT; mn=mn+1;
        end
        if tht<-pi/2, tht=tht+pi/2; elseif tht>pi/2, tht=tht-pi/2; end
        thtk(k+1)=tht;
        if mn>size(su,2), mn=1; end
        if mod(k,2)==1
            yc=sum(yr(:, LBN2:LB));
            th_hat(k)=??*(yc<0);
            % if k>1&th_hat(k-1)<0, th_hat(k)=-th_hat(k); end
        else
            ys=sum(yr(:, LBN2:LB));
            th_hat(k)=????*(2*(ys<0)-1);
        end
        if k>1 % Detector(DTR)
            d=(dth_hat>0); if abs(dth_hat)>=pi, d=~d; end
            nobe = nobe+(d~=s(??)); if nobe>100; break; end
        end
    end
    pobe(iter)= nobe/((k-1)*b);
end
semilogy(SNRbDbt, pbe_PSK, 'k', SNRbDbt, pbe_FSK, 'k:', SNRdBs, pobe, 'b*')

```

Compared with Eq. (7.2.9), which is the error probability for BFSK signaling, the SNR has increased by two times, which is attributed to the doubled integration period $2T_b$ of the correlators.

- (a) Complete the above incomplete program “sim_MSK.m” so that it can simulate the MSK modulation and demodulation process.
 (b) Consider the in-phase/quadrature symbol shaping functions

$$g_c(t) = \begin{cases} \sqrt{\frac{2E_b}{T_b}} \cos\left(\frac{\pi}{2T_b}t\right) & \text{for } -T_b \leq t \leq T_b \\ 0 & \text{elsewhere} \end{cases} \quad (\text{P7.6.12a})$$

$$g_s(t) = \begin{cases} \sqrt{\frac{2E_b}{T_b}} \sin\left(\frac{\pi}{2T_b}t\right) & \text{for } 0 \leq t \leq 2T_b \\ 0 & \text{elsewhere} \end{cases} \quad (\text{P7.6.12b})$$

Verify that their CTFTs and ESDs (energy spectral densities) can be obtained as

$$\begin{aligned} G_c(\omega) &= \mathcal{F}\{g_c(t)\} \stackrel{(1.3.2a)}{=} \sqrt{\frac{2E_b}{T_b}} \int_{-T_b}^{T_b} \cos\left(\frac{\pi}{2T_b}t\right) e^{-j\omega t} dt \\ &= \sqrt{\frac{E_b}{2T_b}} \int_{-T_b}^{T_b} \left(e^{j\pi t/2T_b} + e^{-j\pi t/2T_b}\right) e^{-j\omega t} dt \\ &= \sqrt{\frac{E_b}{2T_b}} \left(\frac{e^{j(\pi/2T_b - \omega)t}}{j(\pi/2T_b - \omega)} - \frac{e^{-j(\pi/2T_b + \omega)t}}{j(\pi/2T_b + \omega)} \right) \Big|_{-T_b}^{T_b} \\ &= \sqrt{\frac{E_b}{2T_b}} \left(\frac{2 \sin(\pi/2T_b - \omega)T_b}{\pi/2T_b - \omega} + \frac{2 \sin(\pi/2T_b + \omega)T_b}{\pi/2T_b + \omega} \right) \\ &= \sqrt{\frac{2E_b}{T_b}} \frac{(\pi/T_b) \sin(\pi/2) \cos \omega T_b - 2\omega \cos(\pi/2) \sin \omega T_b}{(\pi/2T_b)^2 - \omega^2} \\ &= \sqrt{\frac{2E_b}{T_b}} \frac{(\pi/T_b) \cos \omega T_b}{(\pi/2T_b)^2 - \omega^2} \end{aligned} \quad (\text{P7.6.13a})$$

$$\begin{aligned} G_s(\omega) &= \mathcal{F}\{g_s(t)\} \stackrel{(1.3.2a)}{=} \sqrt{\frac{2E_b}{T_b}} \int_0^{2T_b} \sin\left(\frac{\pi}{2T_b}t\right) e^{-j\omega t} dt \\ &= \sqrt{\frac{2E_b}{T_b}} \int_{-T_b}^{T_b} \sin\left(\frac{\pi}{2T_b}(t+T_b)\right) e^{-j\omega(t+T_b)} dt \\ &= \sqrt{\frac{2E_b}{T_b}} \int_{-T_b}^{T_b} \cos\left(\frac{\pi}{2T_b}t\right) e^{-j\omega t} dt e^{-j\omega T_b} \\ &\stackrel{(\text{P7.6.13a})}{=} \sqrt{\frac{2E_b}{T_b}} \frac{(\pi/T_b) \cos \omega T_b}{(\pi/2T_b)^2 - \omega^2} e^{-j\omega T_b} \end{aligned} \quad (\text{P7.6.13b})$$

$$; \Phi_{gc}(\omega) = \Phi_{gs}(\omega) = |G_{gc}(\omega)|^2 = |G_{gs}(\omega)|^2 = \frac{2\pi^2 E_b}{T_b^3} \frac{\cos^2 \omega T_b}{((\pi/2T_b)^2 - \omega^2)^2} \quad (\text{P7.6.14})$$

7.7 Simulation of FSK Using Simulink

Fig. P7.7.1 shows a Simulink model ("FSK_passband_sim.mdl") to simulate a BFSK (binary frequency shift keying) passband communication system and Fig. P7.7.2 shows the subsystem for the demodulator and detector where the following parameters are to be set in the workspace:

```

b=1; M=2^b; % Number of bits per symbol and Modulation order
Ns=40; Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
dw=2*pi/Ts; wc=10*dw; % Frequency spacing and Carrier freq[rad/s]
EbN0dB=10 % Eb/N0 [dB]
Target_no_of_error=50; % Simulation stopping criterion

```

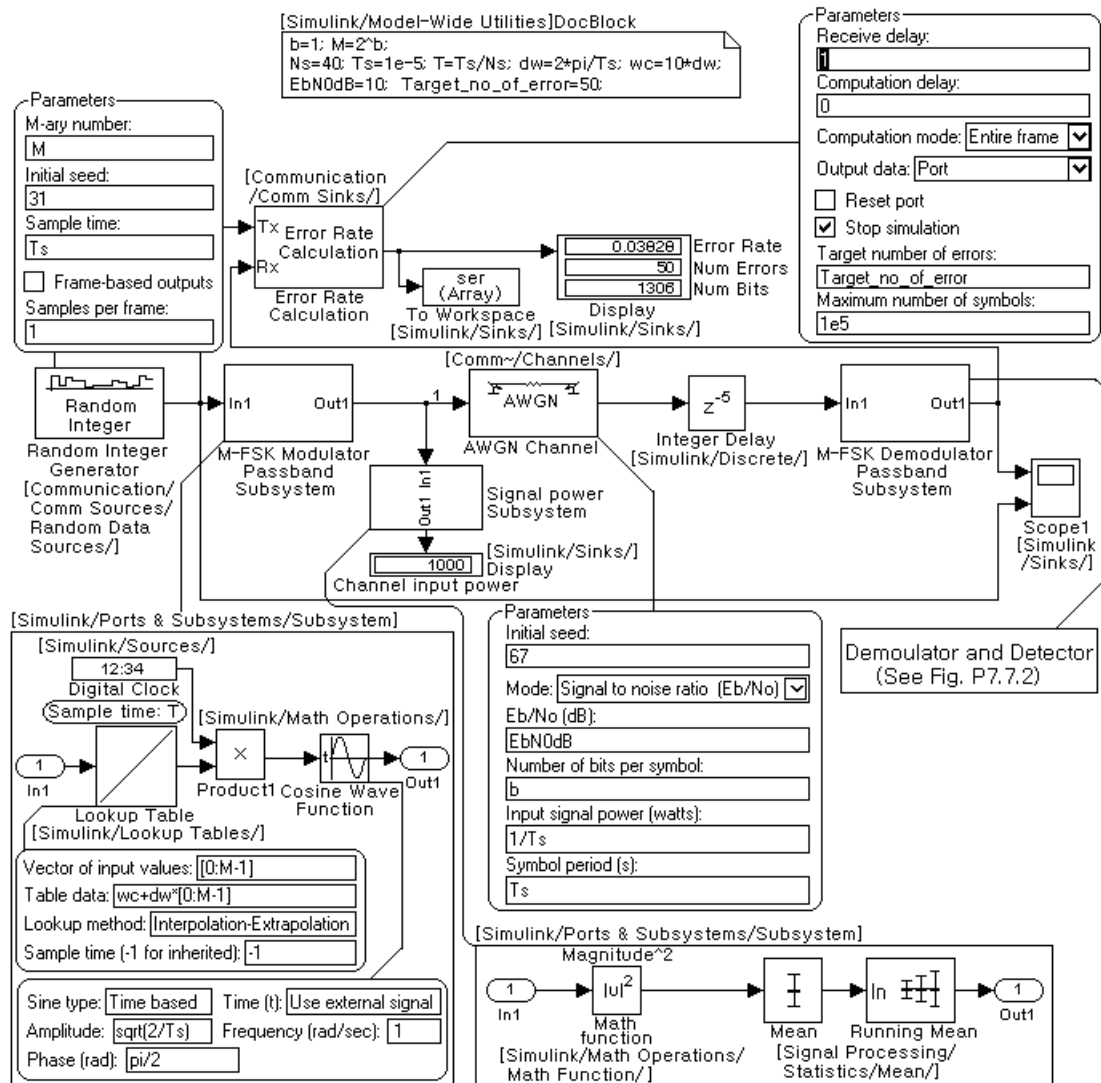


Figure P7.7.1 Simulink model for simulating a noncoherent BFSK communication system ("FSK_passband_sim.mdl")

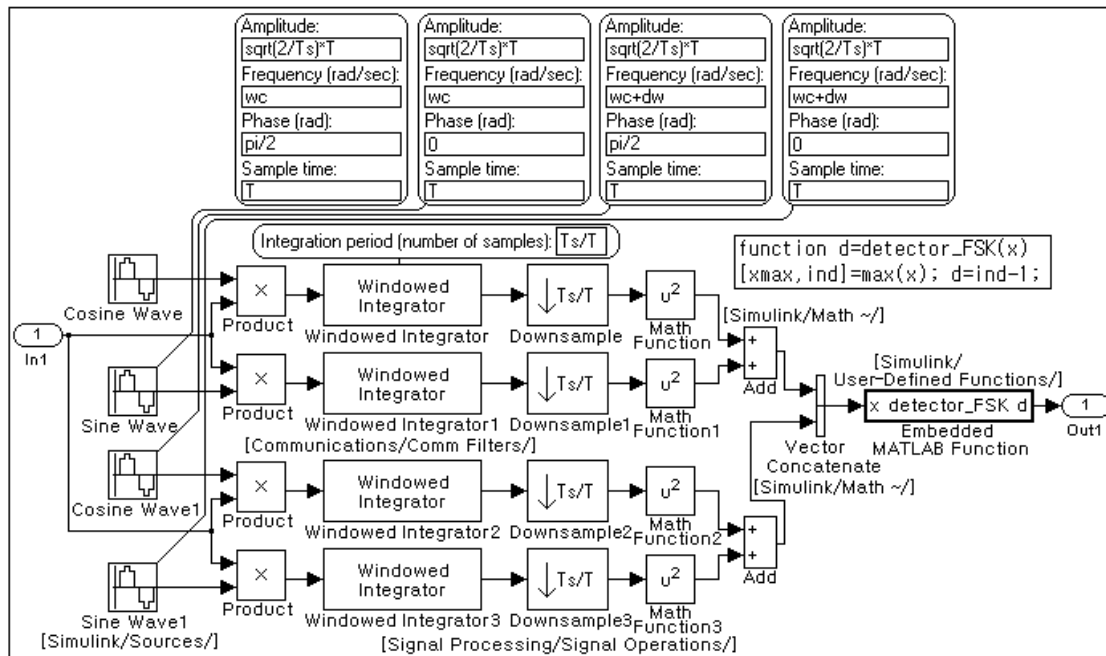


Figure P7.7.2 The subsystem for the demodulator and detector

```

%do_FSK_sim.m
clear, clf
b=1; M=2^b; % Number of bits per symbol and Modulation order
Nbsps=b*2^3; % # of samples per symbol in baseband
Nos=5; Ns=Nbsps*Nos; % # of samples per symbol in passband
Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
dw=2*pi/Ts; % Frequency separation [rad/s]
wc=10*dw; % Carrier Frequency[rad/s] (such that wc*T<pi)
Target_no_of_error=50; SNRdBs=[5 10]; EbN0dBs=SNRdBs-3;
for i=1:length(SNRdBs)
    SNRdB=SNRdBs(i); EbN0dB=SNRdB-3; % Eb/(N0/2)=SNR-> Eb/N0=SNR/2
    sim('FSK_passband_sim',1e5*Ts); SERs(i)=ser(end,1);
end
SNRdBt=0:0.1:13;
poset = prob_error(SNRdBt,'FSK',b,'sym','noncoherent');
semilogy(SNRdBt,poset,'k', SNRdBs,SERs,'*'), xlabel('SNR [dB]')
Transmitted_Signal_Power = 1/Ts;
Received_Signal_Power = ...
    Transmitted_Signal_Power*(1+10^(-(EbN0dBs(end)+3)/10)*Ts/b/T)

function d=detector_FSK(z2s)
[z2max,ind]=max(z2s); d=ind-1; % Fig. 7.4.2

```

Once the Simulink model named “FSK_passband_sim.mdl” is opened, it can be run by clicking on the Run button on the tool bar or by using the MATLAB command ‘sim()’ as in the above program “do_FSK_sim.m”. You can disconnect the Display Block for signal power measurement to increase the running speed.

- (cf) Note that the receive delay in the Error Rate Calculation block is set to 1 due to the sample difference between the transmitted data and detected data observed through the Scope.
- (a) After having composed and saved the Simulink model “FSK_passband_sim.mdl” (with no delay) and two MATLAB programs “do_FSK_sim.m” and “detector_FSK.m” (that can be substituted by an Embedded MATLAB Function Block), run the MATLAB program “do_FSK_sim.m” to get the SERs (SER: symbol error rate) for a BFSK communication system with SNRdBs=[5 10]. Does the simulation result conform with that in Fig. 7.5 (for $b = 1$) or 7.6.2?
- (b) Change the 7th statement of the MATLAB program “do_FSK_sim.m” into ‘dw=pi/Ts’ or ‘dw=3*pi/Ts’ and run the program to get the SER. How are the SERs compared with those obtained in (a)? What does the difference come from?
- (c) Insert the delay of 1~5 samples and discuss the sensitivity of the system w.r.t. such a delay in terms of the SER performance in connection with coherence or noncoherence of the communication system.
- (d) Referring to Fig. 7.4.2, extend the Simulink model into “FSK4_passband_sim.mdl” so that it can simulate an $M=2^2=4$ -ary FSK communication system and run it to get the SER curve.
- (e) To understand how large the power of noise added by the AWGN Block is, connect the Signal power Subsystem together with the Display Block into the input of the AWGN block, click on the Run button, and read the displayed value of the channel input signal power. Then connect the Signal power Subsystem together with the Display Block into the output of the AWGN Block, click on the Run button, and read the displayed value of the channel output signal power that will be the sum of the input signal power and noise power (variance). Do the channel input and output powers measured through the Simulink simulation conform with those obtained using the last two statements in the above program “do_FSK_sim.m”:

```
Transmitted_Signal_Power = 1/Ts;
Received_Signal_Power = ...
    Transmitted_Signal_Power*(1+10^(-(EbN0dBs(end)+3)/10)*Ts/b/T)
```

Why is 3 added to EbN0dBs(end) in the above statement?

- (cf) Visit the webpage <<http://www.mathworks.com/access/helpdesk/help/helpdesk.html>> and refer to the explanation about the function of AWGN Block that can be found in the user manual for Communications Blockset, according to which the SNR is defined to be

$$SNRdB = 10 \log_{10} \frac{\text{Signal Power}}{\text{Noise Variance}}$$

$$= \begin{cases} 10 \log_{10} \frac{E_s/T_s}{N_0/T} = E_s N_0 \text{dB} + 10 \log_{10} \frac{T}{T_s} = E_b N_0 \text{dB} + 10 \log_{10} b \frac{T}{T_s} & \text{for a complex-valued signal} \\ 10 \log_{10} \frac{E_s/T_s}{(N_0/2)T} = E_s N_0 \text{dB} + 10 \log_{10} \frac{2T}{T_s} = E_b N_0 \text{dB} + 10 \log_{10} b \frac{T}{T_s} + 3 & \text{for a real-valued signal} \end{cases} \quad (\text{P7.7.1})$$

$$\frac{\text{Signal Power}}{\text{Noise Variance}} = \begin{cases} (T_s/bT)10^{-EbN0dB/10} & \text{for a complex-valued signal} \\ (T_s/bT)10^{-(EbN0dB+3)/10} & \text{for a real-valued signal} \end{cases} \quad (\text{P7.7.2})$$

7.8 Simulation of PSK Using Simulink

Fig. P7.8.1 shows a Simulink model to simulate a PSK (phase shift keying) passband communication system where the following parameters are to be set in the workspace:

```
b=1; M=2^b; % Number of bits per symbol and Modulation order
Ns=40; Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
wc=2*pi*10/Ts; % Carrier freq[rad/s]
EbN0dB=7; % Eb/N0[dB]
Target_no_of_error=20; % Simulation stopping criterion
```

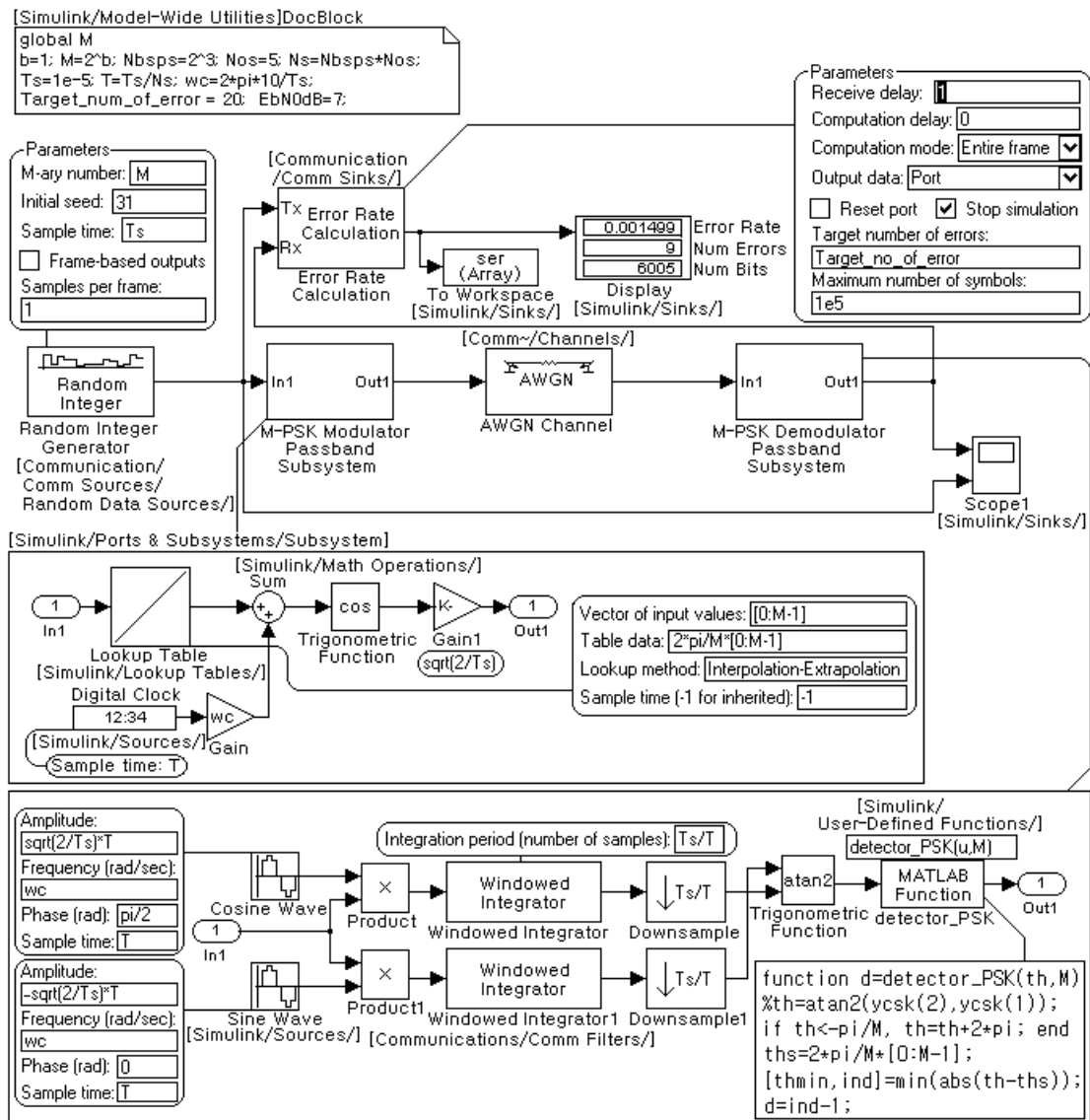


Figure P7.8.1 Simulink model for simulating a PSK communication system ("PSK_passband_sim.mdl")

```

%do_PSK_sim.m
clear, clf
b=1; M=2^b; % Number of bits per symbol and Modulation order
NbSPs=b*2^3; % # of samples per symbol in baseband
Nos=5; Ns=NbSPs*Nos; % # of samples per symbol in passband
Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
wc=2*pi*10/Ts; % Carrier Frequency[rad/s] (such that wc*T<pi)
Target_no_of_error=20;
SNRdBs=[5 10]; EbN0dBs=SNRdBs-3;
for i=1:length(SNRdBs)
    SNRdB=SNRdBs(i); EbN0dB=SNRdB-3; % Eb/(N0/2)=SNR-> Eb/N0=SNR/2
    sim('PSK_passband_sim',1e5*Ts);
    SERs(i) = ser(end,1);
end
SNRdBt=0:0.1:13;
poset = prob_error(SNRdBt,'PSK',b,'sym');
semilogy(SNRdBt,poset,'k', SNRdBs,SERs,'*'), xlabel('SNR[dB]')
Transmitted_Signal_Power = 1/Ts;
Received_Signal_Power = ...
    Transmitted_Signal_Power*(1+10^(-(EbN0dBs(end)+3)/10)*Ts/b/T)

function d=detector_PSK(th,M)
%th=atan2(ycsk(2),ycsk(1));
if th<-pi/M, th=th+2*pi; end
[thmin,ind]=min(abs(th-2*pi/M*[0:M-1])); % Eq.(7.3.9) or Fig. 7.7
d=ind-1;

```

- After having composed and saved the Simulink model “PSK_passband_sim.mdl” and two MATLAB programs “do_PSK_sim.m” and “detector_PSK.m”, run the MATLAB program “do_PSK_sim.m” to get the SERs for a BPSK communication system with $\text{SNRdBs}=[5 \ 10]$. Does the simulation result conform with that in Fig. 7.10 (for $b = 1$)?
- Modify the MATLAB program “do_PSK_sim.m” (possibly together with) Simulink model “PSK_passband_sim.mdl” to simulate an $M=2^2=4$ -ary PSK (QPSK) communication system and run it to get the SER curve.
- Rename the Simulink model “PSK_passband_sim.mdl” “DPSK_passband_sim.mdl” and modify it to simulate an $M=2^2=4$ -ary DPSK (QDPSK) communication system. You can refer to the MATLAB program “sim_DPSK_passband.m” and Fig. P7.8.2. Run it to get the SERs with $\text{SNRdBs}=[5 \ 10]$.

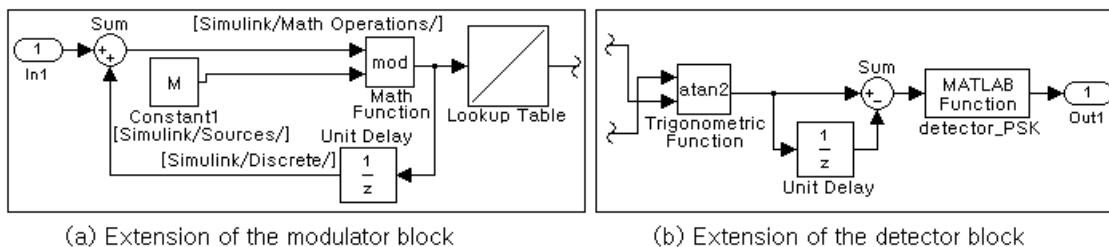


Figure P7.8.2 Modification of the modulator and detector blocks for DPSK

7.9 Simulation of QAM Using Simulink

Fig. P7.9 shows a Simulink model to simulate a QAM (quadrature amplitude modulation) passband communication system where the following parameters are to be set in the workspace:

```
b=4; M=2^b; L=2^(b/2); % Number of bits per symbol and Modulation order
Ns=40; Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
wc=2*pi*10/Ts; % Carrier freq[rad/s]
EbN0dB=10; % Eb/N0 [dB]
Target_no_of_error=100; % Simulation stopping criterion
```

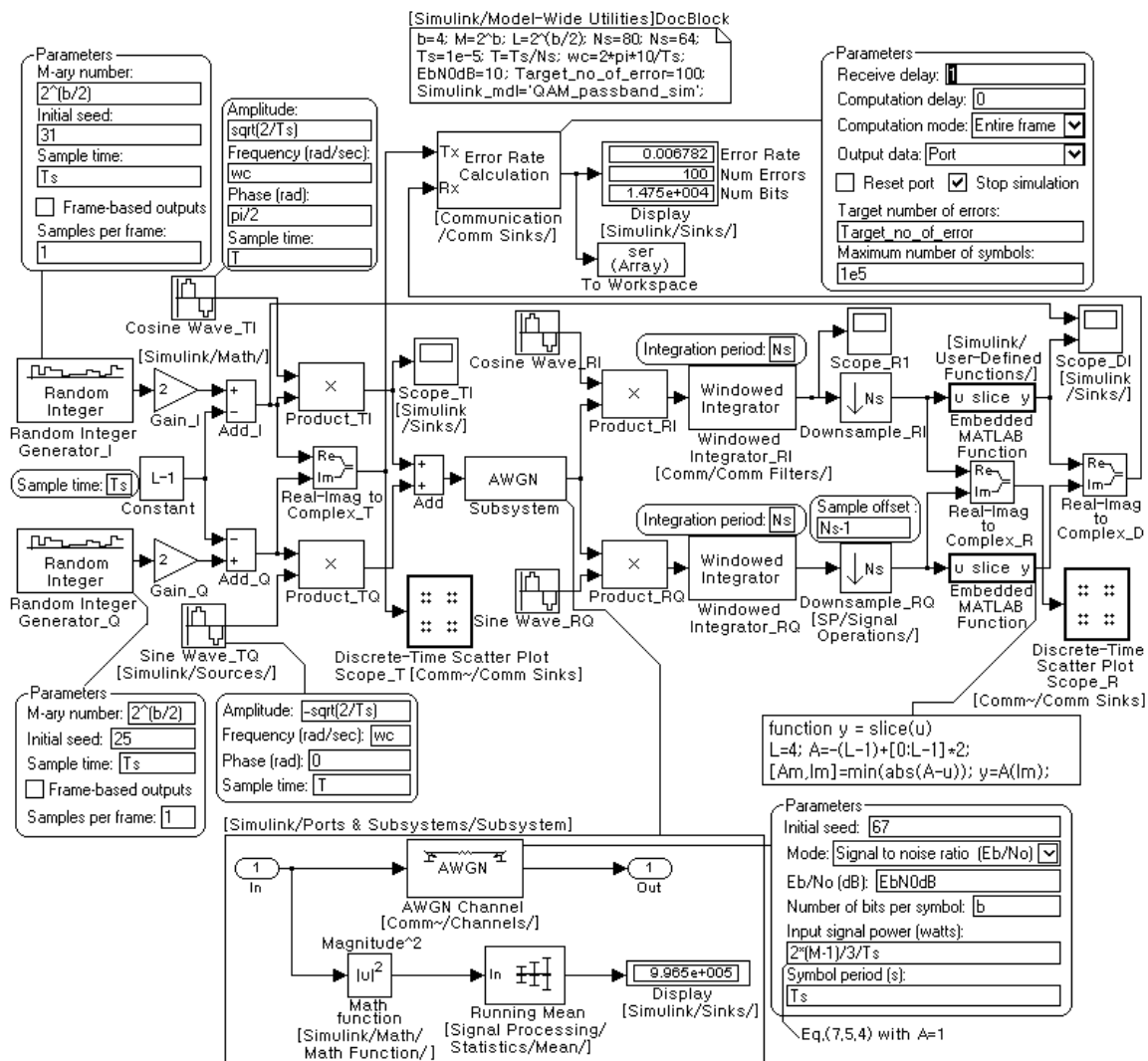


Figure P7.9 Simulink model for simulating a QAM communication system ("QAM_passband_sim.mdl")

```

%do_QAM_sim.m
clear, clf
b=4; M=2^b; L=2^(b/2); % # of bits per symbol and Modulation order
Ns=b*10; Ts=1e-5; T=Ts/Ns; % Symbol time and Sample time
wc=2*pi*10/Ts; % Carrier Frequency[rad/s] (such that wc*T<pi)
Target_no_of_error=100;
SNRdBs=[5 10]; EbN0dBs=SNRdBs-3;
for i=1:length(SNRdBs)
    SNRdB=SNRdBs(i); EbN0dB=SNRdB-3; % Eb/(N0/2)=SNR-> Eb/N0=SNR/2
    sim('QAM_passband_sim',1e5*Ts); SERs(i) = ser(end,1);
end
SNRdBt=0:0.1:13; poset = prob_error(SNRdBt, 'QAM', b, 'sym');
semilogy(SNRdBt, poset, 'k', SNRdBs, SERs, '*'), xlabel('SNR[dB]')
Transmitted_Signal_Power = 2*(M-1)/3/Ts;
Received_Signal_Power = ...
    Transmitted_Signal_Power*(1+10^(-(EbN0dBs(end)+3)/10)*Ts/b/T)

function a=slice(x,A)
[Am,Im]=min(abs(A-x)); a=A(Im);

```

- (a) After having composed and saved the Simulink model “QAM_passband_sim.mdl” and two MATLAB programs “do_QAM_sim.m” and “slice.m” (that can be substituted by an Embedded MATLAB Function Block), run the MATLAB program “do_QAM_sim.m” to get the SERs for SNRdBs=[5 10]. Does the simulation result conform with the theoretical symbol error probability that can be obtained using the MATLAB routine “prob_error()” in Sec. 7.6?
- (b) Modify the program “do_QAM_sim.m” and/or Simulink model “QAM_passband_sim.mdl” to simulate an $M=2^6=64$ -ary QAM communication system and run it to get the SER curve.

```

%do_MSK_sim.m
clear, clf
SNRdBt=[0:0.1:10]; SNRdBs=[5 10]; EbN0dBs= SNRdBs-3;
b=1; M=2^b; % Number of bits per symbol and Modulation order
Tb=1; Nb=16; T=Tb/Nb; Ts=b*Tb; % Bit duration and Sample time
wc=4*pi/Tb; % Carrier frequency[rad/s]
t=[0:2*Nb-1]*T; pihTbt=pi/2/Tb*t;
suT=sqrt(2/Tb)*T*[cos(pihTbt-pi/2).*cos(wc*(t-Tb));
    sin(pihTbt).*sin(wc*t)]; % Eq. (7.6.8a,b)
M_filter1=fliplr(suT(1,:)); M_filter2=fliplr(suT(2,:));
Target_no_of_error = 50;
Simulink_md1='MSK_passband_sim';
for i=1:length(SNRdBs)
    SNRdB = SNRdBs(i); EbN0dB = SNRdB-3;
    sim(Simulink_md1,1e5*Tb), BERs(i) = ber(end,1);
end
pobet_PSK=prob_error(SNRdBt, 'PSK', b, 'bit');
pobet_FSK=prob_error(SNRdBt, 'FSK', b, 'bit');
semilogy(SNRdBt, pobet_PSK, 'k', SNRdBt, pobet_FSK, 'k:', ...
    SNRdBs, BERs, 'b*')

function d=detector_MSK(dth)
if dth>0, d=1; else d=0; end
if abs(dth)>pi, d=1-d; end

```

7.10 Simulation of MSK Using Simulink

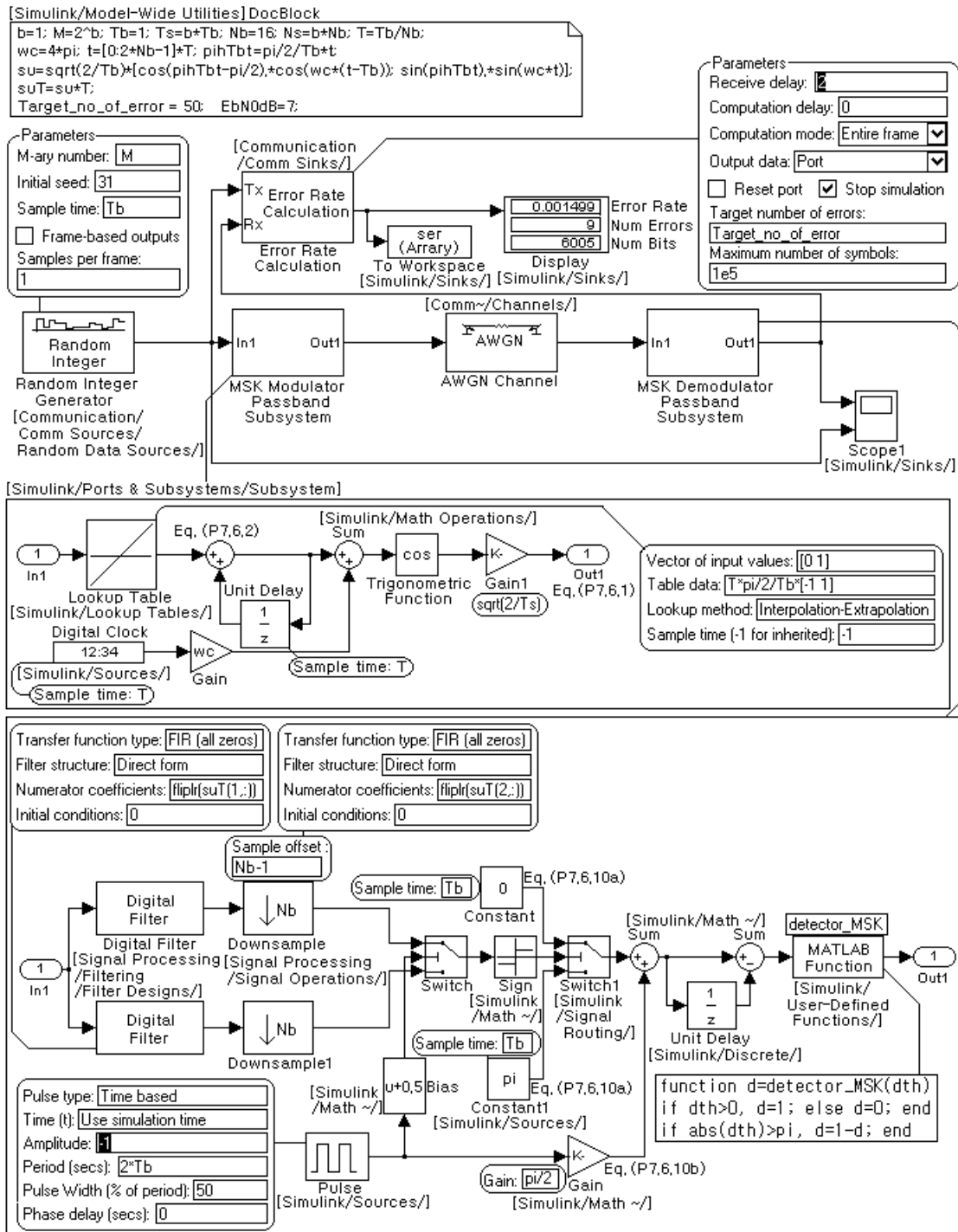


Figure P7.10 Simulink model for simulating an MSK communication system ("MSK_passband_sim.mdl")

Fig. P7.10 shows a Simulink model to simulate an MSK (minimum shift keying) passband communication system where the following parameters are to be set in the workspace. Note that the demodulator is implemented using matched (FIR) filters (whose coefficient vectors or impulse responses are the reversed and delayed versions of the basis signal waveforms (P7.6.8a,b)) instead of correlators. Having composed and saved the Simulink model “MSK_passband_sim.mdl” and two MATLAB programs “do_MSK_sim.m” and “detector_MSK.m”, run the MATLAB program “do_MSK_sim.m” to get the BERs for SNRdBs=[5 10]. Does the simulation result go into between the theoretical symbol error probabilities for FSK and PSK?

```

b=1; M=2^b; % Number of bits per symbol and Modulation order
Nb=16; Ns=b*Nb;
Tb=1e-5; Ts=b*Tb; T=Ts/Ns; % Symbol/Bit time and Sample time
wc=2*pi*10/Tb; % Carrier freq[rad/s]
t=[0:2*Nb-1]*T; pihTbt=pi/2/Tb*t;
suT=sqrt(2/Tb)*T*[cos(pihTbt-pi/2).*cos(wc*(t-Tb));
sin(pihTbt).*sin(wc*t)];
M_filter1=fliplr(suT(1,:)); M_filter2=fliplr(suT(2,:));
EbN0dB=7; % Eb/N0[dB]
Target_no_of_error=20; % Simulation stopping criterion

```


9.4.3 Cyclic Coding

A *cyclic code* is a linear block code having the property that a cyclic shift (rotation) of any codeword yields another codeword. Due to this additional property, the encoding and decoding processes can be implemented more efficiently using a feedback shift register. An (N, K) cyclic code can be described by an $(N - K)$ th-degree generator polynomial

$$\mathbf{g}(x) = g_0 + g_1x + g_2x^2 + \cdots + g_{N-K}x^{N-K} \quad (9.4.24)$$

The procedure of encoding a K -bit message vector $\mathbf{m} = [m_0 \ m_1 \ \cdots \ m_{K-1}]$ represented by a $(K-1)$ th-degree polynomial

$$\mathbf{m}(x) = m_0 + m_1x + m_2x^2 + \cdots + m_{K-1}x^{K-1} \quad (9.4.25)$$

into an N -bit codeword represented by an $(N-1)$ th-degree polynomial is as follows:

1. Divide $x^{N-K}\mathbf{m}(x)$ by the generator polynomial $\mathbf{g}(x)$ to get the remainder polynomial $\mathbf{r}_m(x)$.
2. Subtract the remainder polynomial $\mathbf{r}_m(x)$ from $x^{N-K}\mathbf{m}(x)$ to obtain a codeword polynomial

$$\begin{aligned} \mathbf{c}(x) &= x^{N-K}\mathbf{m}(x) \oplus \mathbf{r}_m(x) = \mathbf{q}(x)\mathbf{g}(x) \\ &= r_0 + r_1x + \cdots + r_{N-K-1}x^{N-K-1} + m_0x^{N-K} + m_1x^{N-K+1} + \cdots + m_{K-1}x^{N-1} \end{aligned} \quad (9.4.26)$$

which has the generator polynomial $\mathbf{g}(x)$ as a (multiplying) factor. Then the first $(N - K)$ coefficients constitute the parity vector and the remaining K coefficients make the message vector. Note that all the operations involved in the polynomial multiplication, division, addition, and subtraction are not the ordinary arithmetic ones, but the modulo-2 operations.

(Example 9.6) A Cyclic Code

With a (7,4) cyclic code represented by the generator matrix

$$\mathbf{g}(x) = g_0 + g_1x + g_2x^2 + g_3x^3 = 1 + 1 \cdot x + 0 \cdot x^2 + 1 \cdot x^3 \quad (\text{E9.6.1})$$

find the codeword for a message vector $\mathbf{m} = [1 \ 0 \ 1 \ 1]$.

Noting that $N = 7$, $K = 4$, and $N - K = 3$, we divide $x^{N-K}\mathbf{m}(x)$ by $\mathbf{g}(x)$ as

$$\begin{aligned} x^{N-K}\mathbf{m}(x) &= x^3(1 + 0 \cdot x + 1 \cdot x^2 + 1 \cdot x^3) = x^6 + x^5 + x^3 = \mathbf{q}(x)\mathbf{g}(x) + \mathbf{r}_m(x) \\ &= (x^3 + x^2 + x + 1)(x^3 + x + 1) + 1 \quad (\text{modulo-2 operation}) \end{aligned} \quad (\text{E9.6.2})$$

to get the remainder polynomial $\mathbf{r}_m(x) = 1$ and add it to $x^{N-K}\mathbf{m}(x) = x^3\mathbf{m}(x)$ to make the codeword polynomial as

$$\begin{aligned} \mathbf{c}(x) &= \mathbf{r}_m(x) + x^3\mathbf{m}(x) = 1 + 0 \cdot x + 0 \cdot x^2 + 1 \cdot x^3 + 0 \cdot x^4 + 1 \cdot x^5 + 1 \cdot x^6 \\ &\rightarrow \mathbf{c} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \\ &\quad \text{parity} \quad | \quad \text{message} \end{aligned} \quad (\text{E9.6.3})$$

The codeword made in this way has the $N - K = 3$ parity bits and $K = 4$ message bits.

Now, let us consider the procedure of decoding a cyclic coded vector. Suppose the RCVR has received a possibly corrupted code vector $\mathbf{r}=\mathbf{c}+\mathbf{e}$ where \mathbf{c} is a codeword and \mathbf{e} is an error. Just as in the encoder, this received vector, being regarded as a polynomial, is divided by the generator polynomial $\mathbf{g}(x)$

$$\mathbf{r}(x) = \mathbf{c}(x) + \mathbf{e}(x) = \mathbf{q}(x)\mathbf{g}(x) + \mathbf{e}(x) = \mathbf{q}'(x)\mathbf{g}(x) + \mathbf{s}(x) \quad (9.4.27)$$

to yield the remainder polynomial $\mathbf{s}(x)$. This remainder polynomial \mathbf{s} may not be the same as the error vector \mathbf{e} , but at least it is supposed to have a crucial information about \mathbf{e} and therefore, may well be called the *syndrome*. The RCVR will find the error pattern \mathbf{e} corresponding to the syndrome \mathbf{s} , subtract it from the received vector \mathbf{r} to get hopefully the correct codeword

$$\mathbf{c} = \mathbf{r} \oplus \mathbf{e} \quad (9.4.28)$$

and accept only the last K bits (in ascending order) of this corrected codeword as a message.

The polynomial operations involved in encoding/decoding every block of message/coded sequence seem to be an unbearable computational load. However, we fortunately have divider circuits which can perform such a modulo-2 polynomial operation. Fig. 9.9 illustrates the two divider circuits (consisting of linear feedback shift registers) each of which carries out the modulo-2 polynomial operations for encoding/decoding with the cyclic code given in Example 9.6. Note that the encoder/decoder circuits process the data sequences in descending order of polynomial.

The encoder/decoder circuits are cast into the MATLAB routines ‘cyclic_encoder()’ and ‘cyclic_decoder()’, respectively, and we make a program “do_cyclic_code.m” that uses the two routines ‘cyclic_encoder()’ and ‘cyclic_decoder()’ (including ‘cyclic_encoder0()’) to simulate the encoding/decoding process with the cyclic code given in Example 9.6. Note a couple of things about the decoding routine ‘cyclic_decoder()’:

- It uses a table of error patterns in the matrix \mathbf{E} , which has every correctable error pattern in its rows. The table is searched for a suitable error pattern by using an error pattern index vector \mathbf{epi} , which is arranged by the decimal-coded syndrome and therefore, can be addressed efficiently by a syndrome just like a decoding hardware circuit.
- If the error pattern table \mathbf{E} and error pattern index vector \mathbf{epi} are not supplied from the calling program, it uses ‘cyclic_decoder0()’ to supply itself with them.

```
% do_cyclic_code.m
% tries with a cyclic code.
clear
N=7; K=4; % N=15; K=7; % Codeword (Block) length and Message size
%N=31; K=16;
g=cyclpoly(N,K); g_=fliplr(g);
lm=5*K; msg= randint(1,lm);
% N=7; K=4; g_=[1 1 0 1]; g=fliplr(g_); msg=[1 0 1 1];
coded = cyclic_encoder(msg,N,K,g_); lc=length(coded);
no_transmitted_bit_errors=ceil(lc*0.05);
errors=randerr(1,lc,no_transmitted_bit_errors);
r = rem(coded+errors,2); % Received sequence
decoded = cyclic_decoder(r,N,K,g_); nobe=sum(decoded~=msg)
coded1 = encode(msg,N,K,'cyclic',g); % Use the Communication Toolbox
r1 = rem(coded1+errors,2); % Received sequence
decoded1 = decode(r1,N,K,'cyclic',g); nobe1=sum(decoded1~=msg)
```

```

function coded= cyclic_encoder(msg_seq,N,K,g)
% Cyclic (N,K) encoding of input msg_seq m with generator polynomial g
Lmsg=length(msg_seq); Nmsg=ceil(Lmsg/K);
Msg= [msg_seq(:); zeros(Nmsg*K-Lmsg,1)];
Msg= reshape(Msg,K,Nmsg).';
coded= [];
for n=1:Nmsg
    msg= Msg(n,:);
    for i=1:N-K, x(i)=0; end
    for k=1:K
        tmp= rem(msg(K+1-k)+x(N-K),2); % msg(K+1-k)+g(N-K+1)*x(N-K)
        for i=N-K:-1:2, x(i)= rem(x(i-1)+g(i)*tmp,2); end
        x(1)=g(1)*tmp;
    end
    coded= [coded x msg]; % Eq.(9.4.26)
end

function [decodes,E,epi]=cyclic_decoder(code_seq,N,K,g,E,epi)
% Cyclic (N,K) decoding of received code_seq with generator polynomial g
% E: Error Pattern matrix or syndromes
% epi: error pattern index vector
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
if nargin<6
    nceb=ceil((N-K)/log2(N+1)); % Number of correctable error bits
    E = combis(N,nceb); % All error patterns
    for i=1:size(E,1)
        syndrome=cyclic_decoder0(E(i,:),N,K,g);
        synd_decimal=bin2deci(syndrome);
        epi(synd_decimal)=i; % Error pattern indices
    end
end
if (size(code_seq,2)==1) code_seq=code_seq.'; end
Lcode= length(code_seq); Ncode= ceil(Lcode/N);
Code_seq= [code_seq(:); zeros(Ncode*N-Lcode,1)];
Code_seq= reshape(Code_seq,N,Ncode).';
decodes=[]; syndromes=[];
for n=1:Ncode
    code= Code_seq(n,:);
    syndrome= cyclic_decoder0(code,N,K,g);
    si= bin2deci(syndrome); % Syndrome index
    if 0<si&si<=length(epi) % Syndrome index to error pattern index
        m=epi(si); if m>0, code=rem(code+E(m,:),2); end % Eq.(9.4.28)
    end
    decodes=[decodes code(N-K+1:N)]; syndromes=[syndromes syndrome];
end
if nargin==2, E=syndromes; end

function x=cyclic_decoder0(r,N,K,g)
% Cyclic (N,K) decoding of an N-bit code r with generator polynomial g
for i=1:N-K, x(i)=r(i+K); end
for n=1:K
    tmp=x(N-K);
    for i=N-K:-1:2, x(i)=rem(x(i-1)+g(i)*tmp,2); end
    x(1)=rem(g(1)*tmp+r(K+1-n),2);
end

```

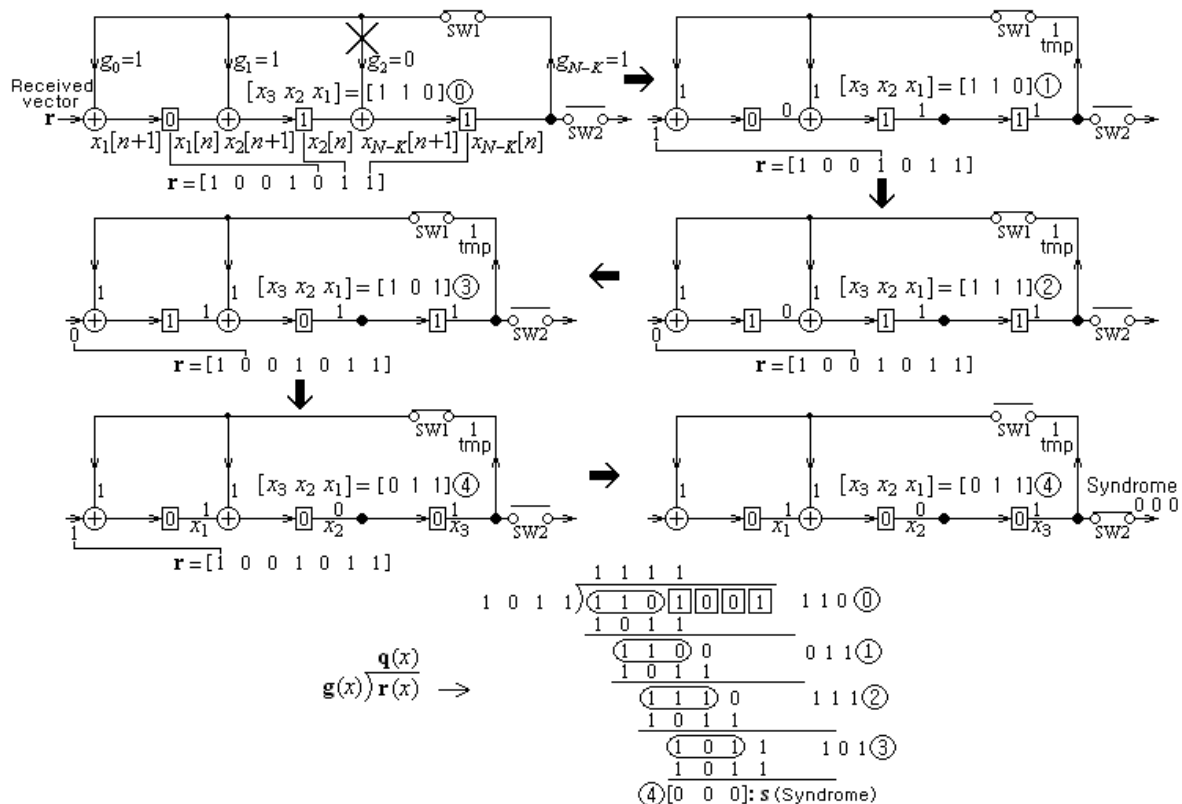
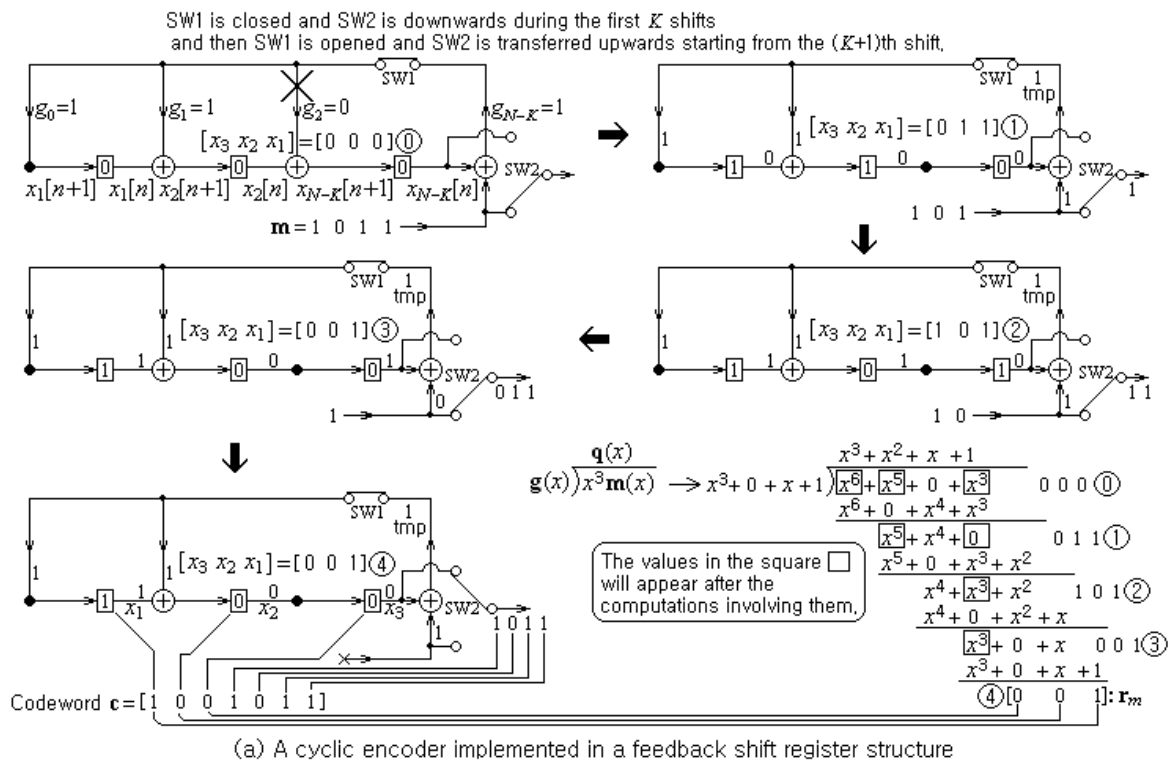


Figure 9.9 An example of cyclic encoding/decoding processes using dividing circuits for polynomial division

Table 9.1 Communication Toolbox functions for block coding

Block coding	Related Communication Toolbox functions and objects
Linear block	encode, decode, gen2par, syndtable
Cyclic	encode, decode, cyclpoly, cyclgen, gen2par, syndtable
BCH (Bose-Chaudhuri-Hocquenghem)	bchenc, bchdec, bchgenpoly
LDPC (Low-Density Parity Check)	fec.ldpcenc, fec.ldpcdec
Hamming	encode, decode, hammggen, gen2par, syndtable
Reed-Solomon	rsenc, rsdec, rsngenpoly, rsencof, rsdecof

Table 9.1 shows a list of several Communication Toolbox functions that can be used to simulate various block coding techniques where the block codings are classified as follows (see the MATLAB Help on Communication Toolbox - block coding):

Linear Block Codes

- LDPC (Low-Density Parity-Check) codes

- Cyclic codes – BCH (Bose-Chadhuri-Hocquenghem) code
 - Hamming codes
 - Reed-Solomon codes

Note the following about the functions 'encode' and 'decode' (use MATLAB Help for details):

- They can be used for any linear block coding by putting a string 'linear' and a $K \times N$ generator matrix as the fourth and fifth input arguments, respectively.
- They can be used for Hamming coding by putting a string 'hamming' as the fourth input argument or by providing them with only the first three input arguments.

The following example illustrates the usages of ‘encode()’/‘decode()’ for cyclic coding and ‘rsenc()’ and ‘rsdec()’ for Reed-Solomon coding. Note that the *RS (Reed-Solomon) codes* are nonbinary BCH codes, which has the largest possible minimum distance for any linear code with the same message size K and codeword length N , yielding the error correcting capability of $d_c=(N-K)/2$.

```
%test_encode_decode.m to try using encode()/decode()
N=7; K=4; % Codeword (Block) length and Message size
g=cyclpoly(N,K); % Generator polynomial for a cyclic (N,K) code
Nm=10; % # of K-bit message vectors
msg=randint(Nm,K); % Nm x K message matrix
coded = encode(msg,N,K,'cyclic',g); % Encoding
% Add bit errors with transmitted BER potbe=0.1
potbe=0.1; received=rem(coded+randerr(Nm,N,[0 1;1-potbe potbe]),2);
decoded=decode(received,N,K,'cyclic',g); % Decoding
% Probability of message bit errors after decoding/correction
pobe=sum(sum(decoded~=msg))/(Nm*K) % BER
% Usage of rsenc()/rsdec()
M=3; % Galois Field integer corresponding to the # of bits per symbol
N=2^M-1; K=3; dc=(N-K)/2; % Codeword length and Message size
msg=gf(randint(Nm,K,2^M),M); % Nm x K GF(2^M) Galois Field msg matrix
coded = rsenc(msg,N,K); % Encoding
noise = randerr(Nm,N,[1 dc+1]).*randint(Nm,N,2^M);
received = coded+noise; % Add a noise
[decoded,numerr]=rsdec(received,N,K); % Decoding
[msg decoded], numerr, pose=sum(sum(decoded~=msg))/(Nm*K) % SER
```

9.4.4 Convolutional Coding and Viterbi Decoding

In the previous sections, we discussed the block coding that encodes every K -bit block of message sequence independently of the previous message block (vector). In this section, we are going to see the *convolutional coding* that converts a K -bit message vector into an N -bit channel input sequence dependently of the previous $(L-1)K$ -bit message vector (L : *constraint length*). The convolutional encoder has a structure of finite-state machine whose output depends on not only the input but also the state.

Fig. 9.10 shows a binary convolutional encoder with a $K(=2)$ -bit input, an $N(=3)$ -bit output, and $L-1(=3)$ 2-bit registers that can be described as a finite-state machine having $2^{(L-1)K}=2^{3 \cdot 2}=64$ states. This encoder shifts the previous contents of every stage register except the right-most one into its righthand one and receives a new K -bit input to load the left-most register at an iteration, sending an N -bit output to the channel for transmission where the values of the output bits depends on the previous inputs stored in the $L-1$ registers as well as the current input.

A binary convolutional code is also characterized by N generator sequences $\mathbf{g}_1, \mathbf{g}_2, \dots, \mathbf{g}_N$ each of which has a length of LK . For example, the convolutional code with the encoder depicted in Fig. 9.10 is represented by the $N(=3)$ generator sequences

$$\begin{aligned} \mathbf{g}_1 &= [0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1] \\ \mathbf{g}_2 &= [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1] \\ \mathbf{g}_3 &= [1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1] \end{aligned} \quad (9.4.29a)$$

which constitutes the generator (polynomial) matrix

$$G_{N \times LK} = \begin{bmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \\ \mathbf{g}_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.4.29b)$$

where the value of the j^{th} element of \mathbf{g}_i is 1 or 0 depending on whether the j^{th} one of the LK bits of the shift register is connected to the i^{th} output combiner or not. The shift register is initialized to all-zero state before the first bit of an input (message) sequence enters the encoder and also finalized to all-zero state by the $(L-1)K$ zero-bits padded onto the tail part of each input sequence. Besides, the length of each input sequence (to be processed at a time) is made to be MK (an integer M times K) even by zero-padding if necessary. For the input sequence made in this way so that its total length is $(M+L-1)K$ including the zeros padded onto it, the length of the output sequence is $(M+L-1)N$ and consequently, the *code rate* will be

$$R_c = \frac{MK}{(M+L-1)N} \xrightarrow{M \rightarrow \infty} \frac{K}{N} \quad \text{for } M \succ L \quad (9.4.30)$$

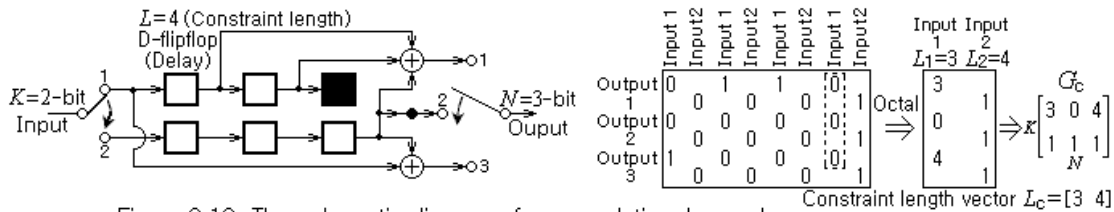


Figure 9.10 The schematic diagram of a convolutional encoder and its representation in the MATLAB functions `convenc()`/`vitdec()`

```

function [output,state]=conv_encoder(G,K,input,state,termmode)
% generates the output sequence of a binary convolutional encoder
% G      : N x LK Generator matrix of a convolutional code
% K      : Number of input bits entering the encoder at each clock cycle.
% input: Binary input sequence
% state: State of the convolutional encoder
% termmode='trunc' for no termination with all-0 state
%Copyleft: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
if isempty(G), output=input; return; end
tmp= rem(length(input),K);
input=[input zeros(1,(K-tmp)*(tmp>0))];
[N,LK]=size(G);
if rem(LK,K)>0
    error('The number of column of G must be a multiple of K!')
end
%L=LK/K;

if nargin<4|(nargin<5 & isnumeric(state))
    input= [input zeros(1,LK)]; %input= [input zeros(1,LK-K)]; end
end
if nargin<4|~isnumeric(state)
    state=zeros(1,LK-K);
end
input_length= length(input);
N_msgsymbol= input_length/K;
input1= reshape(input,K,N_msgsymbol);
output=[];
for l=1:N_msgsymbol % Convolution output=G*input
    ub= input1(:,l)';
    [state,yb]= state_eq(state,ub,G);
    output= [output yb];
end

function [nxb,yb]=state_eq(xb,u,G)
% To be used as a subroutine for conv_encoder()
K=length(u); LK=size(G,2); L1K=LK-K;
if isempty(xb), xb=zeros(1,L1K);
else
    N=length(xb); % (L-1)K
    if L1K~=N, error('Incompatible Dimension in state_eq()'); end
end
A=[zeros(K,L1K); eye(L1K-K) zeros(L1K-K,K)];
B=[eye(K); zeros(L1K-K,K)];
C=G(:,K+1:end); D=G(:,1:K);
nxb=rem(A*xb'+B*u',2)';
yb=rem(C*xb'+D*u',2)';

```

Given a generator matrix $G_{N \times LK}$ together with the number K of input bits and a message sequence \mathbf{m} , the above MATLAB routine ‘conv_encoder()’ pads the input sequence \mathbf{m} with zeros as needed and then generates the output sequence of the convolutional encoder. Communication Toolbox has a convolutional encoding function ‘convenc()’ and its usage will be explained together with that of a convolutional decoding function ‘vitdec()’ at the end of this section.

<Various Representations of a Convolutional Code>

There are many ways of representing a convolutional code such as the schematic diagram like Fig. 9.10, the generator matrix like Eq. (9.4.29b), the finite-state machine (state transition diagram), the transfer function, and the trellis diagram. Fig. 9.11 illustrates the various equivalent representations of a simple convolutional code with $K=1$, $N=2$, and $L=3$. Especially, Fig. 9.11(c1) shows the state diagram that has 4 states corresponding to all possible contents $\{a=00, b=01, c=10, d=11\}$ of the lefthand $(L-1)K$ registers in the schematic diagram (Fig. 9.11(a)) where solid/dashed branches between two states represent the state transitions in the arrow direction in response to input 0/1, respectively. Fig. 9.11(c2) is an augmented state diagram where the all-zero state a in the original state diagram (Fig. 9.11(c1)) is duplicated to make two all-zero states, one (a) with only out-ward branches and the other (a') with only in-ward branches. Note that the self-loop at the all-zero state is neglected so that the gains of all the paths starting from the all-zero state and coming back to the all-zero state can be obtained in a systematic way as follows. If we assign the gain $D^\alpha I^\beta J$ (α : the Hamming weight (number of 1's) of the output sequence assigned to the branch, β : the Hamming weight of the input sequence causing the state change designated by the branch) as in Fig. 9.11(c2), we can regard the state diagram as a signal flow graph and apply the Mason's gain formula^[P-3] to get the overall gain from the input all-zero state a to the output all-zero state a' as

$$\begin{aligned}
 T(D, I, J) &= \frac{1}{\Delta} \sum_i M_i \Delta_i \\
 &= \frac{\sum \text{All products of the forward gain } M_i \text{ and } \Delta_i \text{ for the diagram without forward path } i}{1 - \sum \text{All loop gains} + \sum \text{All products of 2 nontouching loop gains} - \dots} \\
 &= \frac{D^5 I J^3 (1 - DIJ) + D^6 I^2 J^4}{1 - DIJ - DIJ^2 - D^2 I^2 J^3 + D^2 I^2 J^3} = \frac{D^5 I J^3}{1 - (DIJ + DIJ^2)} \quad (9.4.31)
 \end{aligned}$$

This can be expanded into a power series as

$$\begin{aligned}
 T(D, I, J) &= D^5 I J^3 \{1 + (DIJ + DIJ^2) + (DIJ + DIJ^2)^2 + \dots\} \\
 &= D^5 I J^3 + D^6 I^2 J^4 + D^6 I^2 J^5 + D^7 I^3 J^5 + \dots \quad (9.4.32)
 \end{aligned}$$

Each term $D^\alpha I^\beta J^\gamma$ of this equation denotes the gain of each forward path starting from the all-zero state and going back to the all-zero state where α is the Hamming weight of the codeword represented by the path, β is the Hamming weight of the input sequence causing the output sequence to be generated, and γ is the number of branches contained in the path. In this context, the first term $D^5 I J^3$ of Eq. (9.4.32) implies that the convolutional encoder in Fig. 9.11(a) may generate a codeword of Hamming weight $\alpha(5)$ and $\gamma \cdot N = 3 \cdot 2 = 6$ bits for the input sequence of Hamming weight $\beta(1)$:

$$\left. \begin{array}{llll} \text{Input:} & [1] & [0] & [0] \\ \text{State:} & a(00) \rightarrow c(10) \rightarrow b(01) \rightarrow e(00) \\ \text{Output:} & 11 & 01 & 11 \end{array} \right\} \begin{array}{l} \text{--- } \beta=1 \\ \text{--- } \gamma=3 \\ \text{--- } \alpha=5 \end{array} \Rightarrow D^5 I J^3$$

The two codewords corresponding to the terms $D^6 I^2 J^4$ and $D^7 I^3 J^5$ are

$$\left. \begin{array}{llll} \text{Input:} & [1] & [1] & [0] & [0] \\ \text{State:} & a(00) \rightarrow c(10) \rightarrow d(11) \rightarrow b(01) \rightarrow e(00) \\ \text{Output:} & 11 & 10 & 10 & 11 \end{array} \right| \begin{array}{llll} [1] & [1] & [1] & [0] & [0] \\ a(00) \rightarrow c(10) \rightarrow d(11) \rightarrow d(11) \rightarrow b(01) \rightarrow e(00) \\ 11 & 10 & 01 & 10 & 11 \end{array}$$

The minimum degree in D of the transfer function, that is $d_{\min}=5$ in the case of Eq. (9.4.32), is the *free distance* of the convolutional code, which is the *minimum (Hamming) distance* among the codewords in the code. Note that the free distance as well as the decoding algorithm heavily affects the BER performance of a convolutional code.

Fig. 9.11(d) shows another representation of the convolutional code called a *trellis*, which can be viewed as a plot of state diagram being developed along the time. This diagram consists of the columns of $2^{(L-1)K}$ state nodes per clock cycle and the solid/dashed branches representing the state transition caused by input 0/1 as depicted in the state diagram (Fig. 9.11(c)) where the $N(2)$ -bit output of the encoder to the input is attached to each branch. In this trellis diagram, each path starting from the initial all-zero state and ending at another all-zero state represents a codeword. From the trellis shown in Fig. 9.11, we can see the shortest two codewords as

Input:	[1]	[0]	[0]
State:	$a(00) \rightarrow c(10) \rightarrow b(01) \rightarrow a(00)$		
Output:	11	01	11

Input:	[1]	[1]	[0]	[0]
State:	$a(00) \rightarrow c(10) \rightarrow d(11) \rightarrow b(01) \rightarrow a(00)$			
Output:	11	10	10	11

These correspond to the first two terms $D^5 I J^3$ and $D^6 I^2 J^4$ of Eq. (9.4.32), respectively.

<Viterbi Decoding of a Convolutional Coded Sequence>

As a way of decoding a convolutional coded sequence, an ML (maximum-likelihood) decoding called the *Viterbi algorithm* (VA)[F-1] is most widely used where an ML decoding finds the code sequence that is most likely to have been the input to the encoder at XMTR yielding the received code sequence. Given a received (DTR output or decoder input) sequence $\mathbf{r} = \mathbf{m}_c + \mathbf{e}$ (\mathbf{m}_c : a convolutional coded sequence, \mathbf{e} : an error), the *Viterbi algorithm* with *soft-decision* searches the trellis diagram for an all-zero-state-to-all-zero-state path whose (encoder) output sequence is closest to \mathbf{r} in terms of Euclidean distance and takes the encoder input sequence corresponding to the 'optimal' path to be the most likely message sequence, while the Viterbi algorithm with *hard-decision* first slices \mathbf{r} to make \mathbf{r}_q consisting of 0 or 1 and then finds an all-zero-state-to-all-zero-state path whose (encoder) output sequence is closest to \mathbf{r}_q in terms of Hamming distance.

The procedure of the Viterbi algorithm is as follows (see Fig. 9.12):

0. To each set of nodes (states), assign the depth level index starting from $l=0$ for the leftmost stage. Divide the decoder input sequence \mathbf{r}_q (hard, i.e. sliced to 0/1) or \mathbf{r} (soft) into N -bit subsequences and distribute each N -bit subsequence to the stages between the node sets.
1. To each branch, attach the (Hamming or Euclidean) distance between the N -bit encoder output and the N -bit subsequence of \mathbf{r} or \mathbf{r}_q (distributed to the stage that the branch belongs to) as the *branch cost (metric)*.
2. With the level index initialized to $l=0$, assign 0 to the initial all-zero state node as its *node cost (metric)*.
3. Move right by one stage by increasing the level index by one. To every node (at the level) that is connected via some branch(es) to node(s) at the previous level, assign the node cost that is computed by adding the branch cost(s) to the left-hand node cost(s) and taking the minimum if there are multiple branches connected to the node. In the case of multiple branches connected to a node, remove other branches than the survivor branch yielding the minimum path cost.
4. If all the N -bit subsequence of \mathbf{r} or \mathbf{r}_q are not processed, go back to step 3 to repeat the procedure; otherwise, go to the next step 5.
5. Starting from the final all-zero state or the best node with minimum cost, find the optimal path consisting of only the survivor paths and take the encoder input sequence supposedly having caused the output sequence (associated with the optimal path) to be the ML decoded message sequence.

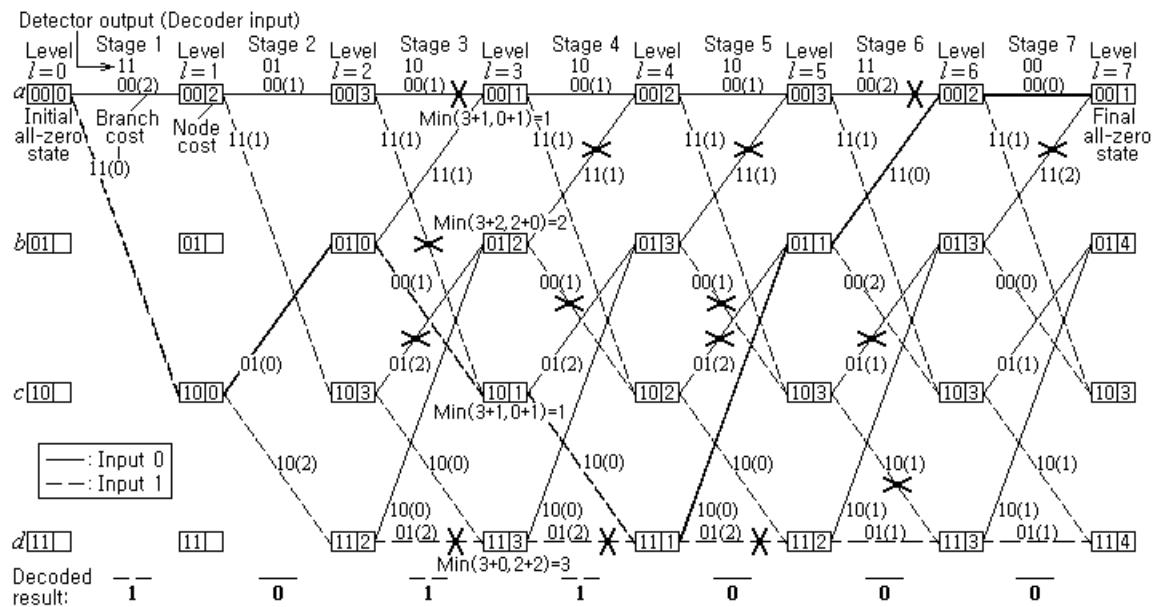


Figure 9.12 The trellis diagram and Viterbi decoding

Fig. 9.12 illustrates a typical decoding procedure using the trellis based on the Viterbi algorithm (VA) stated above where the detector output (code) sequence is [11011010101100]. Note that the removed branches other than the survivor branches are crossed and the optimal path denoted by a series of thick solid/dotted lines (with encoder input 0/1) corresponds to the encoder input sequence [1011000], which has been taken as the decoded result.

Given the generator polynomial matrix G together with the number K of input bits and the channel-DTR output sequence ‘detected’ as its input arguments, the MATLAB routine ‘vit_decoder($G, K, \text{detected}$)’ constructs the trellis diagram and applies the Viterbi algorithm to find the maximum-likelihood decoded message sequence. The following MATLAB program ‘do_vitdecoder.m’ uses the routine ‘conv_encoder()’ to make a convolutional coded sequence for a message and uses ‘vit_decoder()’ to decode it to recover the original message.

```
%do_vitdecoder.m
% Try using conv_encoder()/vit_decoder()
clear, clf
msg=[1 0 1 1 0 0 0]; % msg=randint(1,100)
lm=length(msg); % Message and its length
G=[1 0 1;1 1 1]; % N x LK Generator polynomial matrix
K=1; N=size(G,1); % Size of encoder input/output
potbe=0.02; % Probability of transmitted bit error
% Use of conv_encoder()/vit_decoder()
ch_input=conv_encoder(G,K,msg) % Self-made convolutional encoder
notbe=ceil(potbe*length(ch_input));
error_bits=randerr(1,length(ch_input),notbe);
detected= rem(ch_input+error_bits,2); % Received/modulated/detected
decoded= vit_decoder(G,K,detected)
noe_vit_decoder=sum(msg~=decoded(1:lm))
```

```

function decoded_seq=vit_decoder(G,K,detected,opmode,hard_or_soft)
% performs the Viterbi algorithm on detected to get the decoded_seq
% G: N x LK Generator polynomial matrix
% K: Number of encoder input bits
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
detected = detected(:).';
if nargin<5|hard_or_soft(1)=='h', detected=(detected>0.5); end
[N,LK]=size(G);
if rem(LK,K)~=0, error('Column size of G must be a multiple of K'); end
tmp= rem(length(detected),N);
if tmp>0, detected=[detected zeros(1,N-tmp)]; end
b=LK-K; % Number of bits representing the state
no_of_states=2^b; N_msgsymbol=length(detected)/N;
for m=1:no_of_states
    for n=1:N_msgsymbol+1
        states(m,n)=0; % inactive in the trellis
        p_state(m,n)=0; n_state(m,n)=0; input(m,n)=0;
    end
end
states(1,1)=1; % make the initial state active
cost(1,1)=0; K2=2^K;
for n=1:N_msgsymbol
    y=detected((n-1)*N+1:n*N); % Received sequence
    n1=n+1;
    for m=1:no_of_states
        if states(m,n)==1 % active
            xb=dec2bin1(m-1,b);
            for m0=1:K2
                u=dec2bin1(m0-1,K);
                [nxb(m0,:),yb(m0,:)]=state_eq(xb,u,G);
                nxm0=bin2dec1(nxb(m0,:))+1;
                states(nxm0,n1)=1;
                dif=sum(abs(y-yb(m0,:)));
                d(m0)=cost(m,n)+dif;
                if p_state(nxm0,n1)==0 % Unchecked state node?
                    cost(nxm0,n1)=d(m0);
                    p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1;
                else
                    [cost(nxm0,n1),i]=min([d(m0) cost(nxm0,n1)]);
                    if i==1, p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1; end
                end
            end
        end
    end
end
decoded_seq=[];
if nargin>3 & ~strcmp(opmode,'term',4)
    [min_dist,m]=min(cost(:,n1)); % Trace back from best-metric state
else m=1; % Trace back from the all-0 state
end
for n=n1:-1:2
    decoded_seq= [dec2bin1(input(m,n),K) decoded_seq];
    m=p_state(m,n);
end

```

The following program “do_vitdecoder1.m” uses the Communication Toolbox functions ‘convenc()’ and ‘vitdec()’ where ‘vitdec()’ is used several times with different input argument values to show the readers its various usages. Now, it is time to see the usage of the function ‘vitdec()’.

```
%do_vitdecoder1.m
% shows various uses of Communication Toolbox function convenc()
% with KxN Code generator matrix Gc - octal polynomial representation
clear, clf
%msg=[1 0 1 1 0 0 0];
msg=randint(1,100)
lm=length(msg); % Message and its length
potbe=0.02; % Probability of transmitted bit error
Gc=[5 7]; % 1 0 1 -> 5, 1 1 1 -> 7 (octal number)
Lc=3; % 1xK constraint length vector for each input stream
[K,N]=size(Gc); % Number of encoder input/output bits
trel=poly2trellis(Lc,Gc); % Trellis structure
ch_input1=convenc(msg,trel); % Convolutional encoder
notbel=ceil(potbe*length(ch_input1));
error_bits1=randerr(1,length(ch_input1),notbel);
detected1= rem(ch_input1+error_bits1,2); % Received/modulated/detected
% with hard decision
Tbdepth=3; % Traceback depth
decoded1= vitdec(detected1,trel,Tbdepth, 'trunc', 'hard')
noe_vitdec_trunc_hard=sum(msg~=decoded1(1:lm))
decoded2= vitdec(detected1,trel,Tbdepth, 'cont', 'hard');
noe_vitdec_cont_hard=sum(msg(1:end-Tbdepth)~=decoded2(Tbdepth+1:end))
% with soft decision
ncode= [detected1+0.1*randn(1,length(detected1)) zeros(1,Tbdepth*N)];
quant_levels=[0.001,.1,.3,.5,.7,.9,.999];
NSDB=ceil(log2(length(quant_levels))); % Number of Soft Decision Bits
qcode= quantiz(ncode,quant_levels); % Quantized
decoded3= vitdec(qcode,trel,Tbdepth, 'trunc', 'soft', NSDB);
noe_vitdec_trunc_soft=sum(msg~=decoded3(1:lm))
decoded4= vitdec(qcode,trel,Tbdepth, 'cont', 'soft', NSDB);
noe_vitdec_cont_soft=sum(msg~=decoded4(Tbdepth+1:end))
% Repetitive use of vitdec() to process the data block by block
delay=Tbdepth*K; % Decoding delay depending on the traceback depth
% Initialize the message sequence, decoded sequence,
% state metric, traceback state/input, and encoder state.
msg_seq=[]; decoded_seq=[];
m=[]; s=[]; in=[]; encoder_state=[];
N_Iter=100;
for itr=1:N_Iter
    msg=randint(1,1000); % Generate the message sequence in a random way
    msg_seq= [msg_seq msg]; % Accumulate the message sequence
    if itr==N_Iter, msg=[msg zeros(1,delay)]; end % Append with zeros
    [coded,encoder_state]=convenc(msg,trel,encoder_state);
    [decoded,m,s,in]=vitdec(coded,trel,Tbdepth, 'cont', 'hard', m,s,in);
    decoded_seq=[decoded_seq decoded];
end
lm=length(msg_seq);
noe_repeated_use=sum(msg_seq(1:lm)~=decoded_seq(delay+[1:lm]))
```

<Usage of the Viterbi Decoding Function 'vitdec()' with 'convenc()' and 'poly2trellis()>

To apply the MATLAB functions 'convenc()'/ 'vitdec()', we should first use 'poly2trellis()' to build the trellis structure with an 'octal code generator' describing the connections among the inputs, registers, and outputs. Fig. 9.13 illustrates how the octal code generator matrix G_c as well as the binary generator matrix G and the constraint length vector L_c is constructed for a given convolutional encoder. An example of using 'poly2trellis()' to build the trellis structure for 'convenc()'/ 'vitdec()' is as follows:

```
trellis=poly2trellis(Lc,Gc);
```

Here is a brief introduction of the usages of the Communication Toolbox functions 'convenc()' and 'vitdec()'. See the MATLAB Help manual or The Mathworks webpage[W-7] for more details.

(1) coded=convenc(msg,trellis);

msg: A message sequence to be encoded with a convolutional encoder described by 'trellis'.

(2) decoded=vitdec(coded,trellis,tbdepth,opmode,dectype,NSDB);

coded : A convolutional coded sequence possibly corrupted by a noise. It should consist of binary numbers (0/1), real numbers between 1(logical zero) and -1(logical one), or integers between 0 and $2^{NSDB}-1$ (NSDB: the number of soft-decision bits given as the optional 6th input argument) corresponding to the quantization level depending on which one of {'hard', 'unquant', 'soft'} is given as the value of the fifth input argument 'dectype' (decision type).

trellis : A trellis structure built using the MATLAB function 'poly2trellis()'.

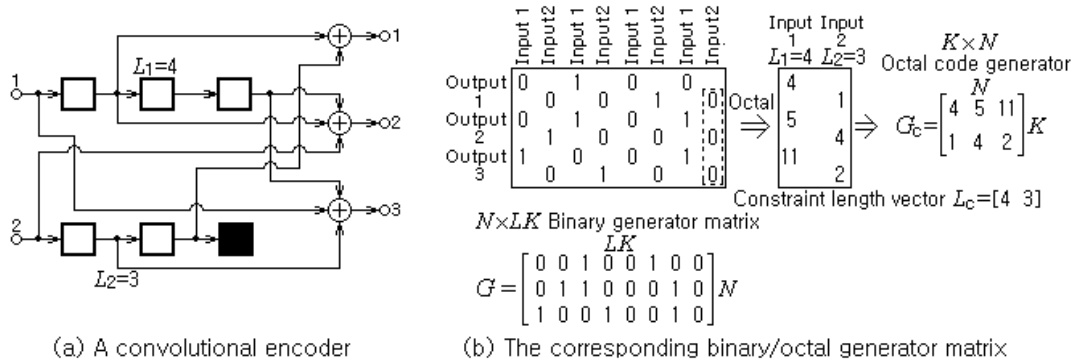
tbdepth: Traceback depth (length), i.e., the number of trellis branches used to construct each traceback path. It should be given as a positive integer, say, about five times the constraint length. In case the fourth input argument 'opmode' (operation mode) is 'cont' (continuous), it causes the decoding delay, i.e., the number of zero symbols preceding the first decoded symbol in the output 'decoded' and as a consequence, the decoded result should be advanced by tbdepth*K where K is the number of encoder input bits.

opmode: Operation mode of the decoding process. If it is set to 'cont' (continuous mode), the internal state of the decoder will be saved for use with the next frame. If it is set to 'trunc' (truncation mode), each frame will be processed independently, and the traceback path starts at the best-metric state and always ends in the all-zero state. If it is set to 'term' (termination mode), each frame is treated independently, and the traceback path always starts and ends in the all-zero state. This mode is appropriate when the uncoded message signal has enough zeros, say, $K * \text{Max}(L_c)-1$ zeros at the end of each frame to fill all memory registers of the encoder.

dectype: Decision type. It should be set to 'unquant', 'hard', or 'soft' depending on the characteristic of the input coded sequence (coded) as follows:

- 'hard' (decision) when the coded sequence consists of binary numbers 0 or 1.
- 'unquant' when the coded sequence consists of real numbers between -1(logical 1) and +1(logical 0).
- 'soft' (decision) when the optional 6th input argument NSDB is given and the coded sequence consists of integers between 0 and $2^{NSDB}-1$ corresponding to the quantization level.

NSDB : Number of software decision bits used to represent the input coded sequence. It is needed and active only when dectype is set to 'soft'.



(3) `[decoded,m,s,in]=vitdec(code,trellis,tbdepth,opmode,dectype,m,s,in)`

This format is used for a repetitive use of 'vitdec()' with the continuous operation mode where the state metric 'm', traceback state 's', and traceback input 'in' are supposed to be initialized to empty sets at first and then handed over successively to the next iteration.

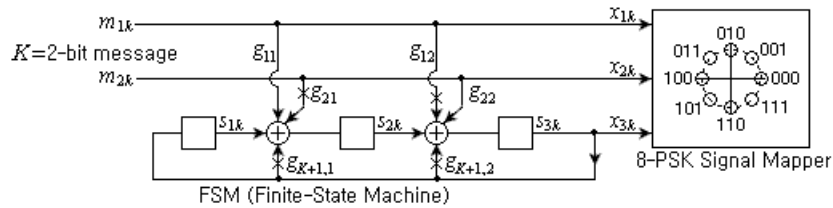
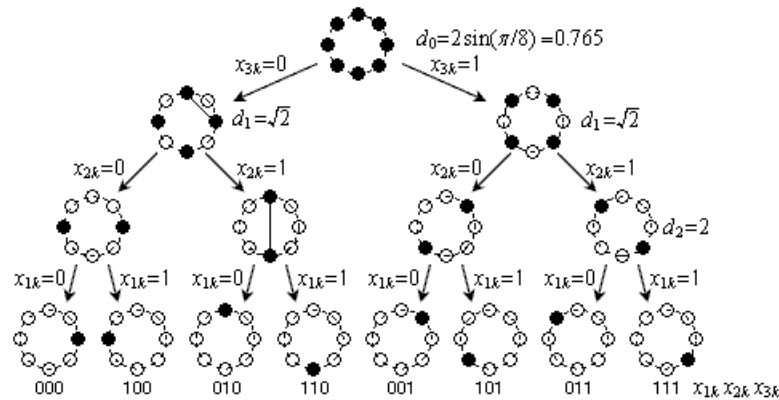
See the above program "do_vitdecoder1.m" or the MATLAB help manual for some examples of using 'convenc()' and 'vitdec()'.

9.4.5 Trellis-Coded Modulation (TCM)

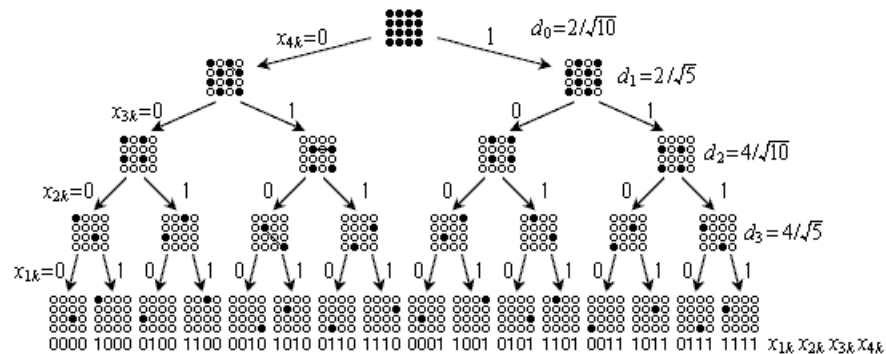
By channel coding discussed in the previous sections, additional redundant bits for error control are transmitted in the code and as a consequence, wider bandwidth is needed to keep the same data rate while lower BER can be obtained with the same SNR or the SNR to obtain the same BER can be decreased. This shows a trade-off between bandwidth efficiency and power efficiency. As an approach to mediate between these two conflicting factors, TCM (trellis-coded modulation) was invented by Gottfried Ungerboeck^[U-1, U-2]. It is just like a marriage between modulation residing on the constellation and coding living on the trellis toward a more effective utilization of bandwidth and power. A (code) rate-2/3 TCM encoder combining an $(N, K) = (3, 2)$ convolutional coding and an $2^3=8$ -PSK modulation is depicted in Fig. 9.14(a). In this encoder, the 8-PSK signal mapper generates one of the eight PSK signals depending on the 3-bit symbol $\mathbf{x}_k = [x_{1k} \ x_{2k} \ x_{3k}]$ that consists of two (uncoded) message bits x_{1k} and x_{2k} and an additional coded bit that is the output of the FSM (finite-state machine). Since the encoding scheme can be described by a trellis, the Viterbi algorithm can be used to decode the TCM coded signal. There are a couple of things to mention about TCM:

- The number of signal points in the constellation is larger compared with the uncoded case. For example, the constellation size or modulation order used by the TCM scheme of Fig. 9.14(a) is $2^{K+1}=8$. This is two times as large as that of QPSK modulation that would be used to send the message by $K=2$ -bit symbols with no coding. Note that a larger modulation order with the same SNR decreases the minimum distance among the codewords so that the BER may suffer.
- TCM has some measures to combat the possible BER performance degradation problem:
 - The convolutional coding allows only certain sequences (paths) of signal points so that the free distance among different signal paths in the trellis can be increased.

- The TCM decoder uses soft-decision decoding to find the path with minimum (squared) Euclidean distance through the trellis. This makes the trellis code design trying to maximize the Euclidean distance among the codewords.
- As illustrated in Fig. 9.14(b1) and (b2), the *set partitioning* let the more significant message bit(s) have a larger Euclidean distance from its complement so that there are less likely errors in the uncoded message bits than in the coded bit(s).
- The FSM of TCM encoder has some inputs added to the adders between the shift registers. Therefore it seems that TCM encoding/decoding can not be implemented using the general convolutional encoder/decoder, requiring its own encoder/decoder routines of every TCM encoder/decoder.

(a) TCM encoder consisting of a 2^3 -state FSM and an 8-PSK signal mapper

(b1) Set partitioning of an 8-PSK signal set into subsets with increasing minimum distance



(b2) Set partitioning of a 16-QAM signal set into subsets with increasing minimum distance

Figure 9.14 An example of TCM (Trellis-Coded Modulation) coding

The following program “sim_TCM.m” uses the two subroutines “TCM_encoder()” and “TCM_decoder()” to simulate the TCM encoding (shown in Fig. 9.14(a)) and the corresponding TCM decoding.

```
%sim_TCM.m
% simulates a Trellis-coded Modulation
clear, clf
lm=1e4; msg=randint(1,lm);
K=2; N=K+1; Ns=3; % Size of encoder input/output/state
M=2^N; Constellation=exp(j*2*pi/M*[0:M-1]); % Constellation
ch_input=TCM_encoder('TCM_state_eq0',K,Ns,N,msg,Constellation);
lc= length(ch_input);
SNRbdB=5; SNRdB=SNRbdB+10*log10(K); % SNR per K-bit symbol
sigma=1/sqrt(10^(SNRdB/10));
noise=sigma/sqrt(2)*(randn(1,lc)+j*randn(1,lc)); % Complex noise
received = ch_input + noise; var(received)
received = awgn(ch_input,SNRdB); var(received) % Alternative
decoded_seq=TCM_decoder('TCM_state_eq0',K,Ns,received,Constellation);
ber_TCM8PSK = sum(decoded_seq(1:lm)~=msg)/lm
ber_QPSK_theory = prob_error(SNRbdB,'PSK',K,'BER')

function [output,state]
    =TCM_encoder(state_eq,K,Ns,N,input,state,Constellation,opmode)
% Generates the output sequence of a binary TCM encoder
% Input: state_eq = External function for state eqn saved in an M-file
%         K       = Number of input bits entering the encoder at each cycle
%         Nsb      = Number of state bits of the TCM encoder
%         N        = Number of output bits of the TCM encoder
%         input=    Binary input message seq.
%         state=    State of the conv_encoder
%         Constellation= Signal sets for signal mapper
% Output: output=   Sequence of signal points on Constellation
%         state =   Updated state
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
tmp= rem(length(input),K);
input= [input zeros(1,(K-tmp)*(tmp>0))];
if nargin<6, state=zeros(1,Nsb);
elseif length(state)==2^N, Constellation=state; state=zeros(1,Nsb);
end
input_length= length(input); N_msgsymbol= input_length/K;
input1= reshape(input,K,N_msgsymbol).';
outputs= [];
for l=1:N_msgsymbol
    ub= input1(l,:);
    [state,output]=feval(state_eq,state,ub,Constellation);
    outputs= [outputs output];
end

function [s1,x]=TCM_state_eq0(s,u,Constellation)
% State equation for the TCM_encoder in Fig. 9.14(a)
% Input: s= State, u= Input, Constellation
% Output: s1= Next state, x= Output
s1= [s(3) rem([s(1)+u(1) s(2)+u(2)],2)];
x= Constellation(bin2deci([u(1) u(2) s(3)]+1));
```

```

function decoded_seq
    =TCM_decoder(state_eq,K,Nsb,received,Constellation,opmode)
% Performs the Viterbi algorithm on the PSK demodulated signal
% Input: state_eq = External function for state eqn saved in an M-file
%         K      = Number of input bits entering the encoder at each cycle.
%         Nsb     = Number of state bits of the TCM encoder
%         received = received sequence
%         Constellation: Signal sets for signal mapper
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
N_states=2^Nsb;
N_msgsymbol=length(received);
for m=1:N_states
    for n=1:N_msgsymbol+1
        states(m,n)=0; %inactive in the trellis diagram
        p_state(m,n)=0; n_state(m,n)=0; input(m,n)=0;
    end
end
states(1,1)=1; % make the initial state active
cost(1,1)=0; K2=2^K;
for n=1:N_msgsymbol
    y=received(n); %received sequence
    n1=n+1;
    for m=1:N_states
        if states(m,n)==1 %active
            xb=dec2bin1(m-1,Nsb);
            for m0=1:K2
                u=dec2bin1(m0-1,K);
                [nxb(m0,:),yb(m0)]=feval(state_eq,xb,u,Constellation);
                nxm0=bin2dec1(nxb(m0,:))+1; states(nxm0,n1)=1;
                % Accumulated squared Euclidean distance as path-to-node cost
                difference=y-yb(m0);
                d(m0)=cost(m,n)+difference*conj(difference);
                if p_state(nxm0,n1)==0
                    cost(nxm0,n1)=d(m0);
                    p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1;
                else
                    [cost(nxm0,n1),i]=min([d(m0) cost(nxm0,n1)]);
                    if i==1, p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1; end
                end
            end
        end
    end
end
decoded_seq=[];
if nargin<6 | ~strcmp(opmode,'term',4)
    % trace back from the best-metric state (default)
    [min_cost,m]=min(cost(:,n1));
else m=1; % trace back from the all-0 state
end
for n=n1:-1:2
    decoded_seq= [dec2bin1(input(m,n),K) decoded_seq];
    m=p_state(m,n);
end

```

9.4.6 Turbo Coding

In order for a linear block code or a convolutional code to approach the theoretical limit imposed by Shannon's channel capacity (see Eq. (9.3.16) or Fig. 9.7) in terms of bandwidth/power efficiency, its codeword or constraint length should be increased to such an intolerable degree that the maximum likelihood decoding can become unrealizable. Possible solutions to this dilemma are two classes of powerful error correcting codes, each called turbo codes and LDPC (lower-density parity-check) codes, that can achieve a near-capacity (or near-Shannon-limit) performance with a reasonable complexity of decoder. The former is the topic of this section and the latter will be introduced in the next section.

Fig. 9.15(a) shows a turbo encoder consisting of two *recursive systematic convolutional* (RSC) encoders and an interleaver where the interleaver permutes the message bits in a random way before input to the second encoder. (Note that the modifier '*systematic*' means that the uncoded message bits are imbedded in the encoder output stream as they are.) The code rate will be 1/2 or 1/3 depending on whether the puncturing is performed or not. (Note that puncturing is to omit transmitting some coded bits for the purpose of increasing the code rate beyond that resulting from the basic structure of the encoder.) Fig. 9.15(b) shows a demultiplexer, which classifies the coded bits into two groups, one from encoder 1 and the other from encoder 2, and applies each of them to the corresponding decoder. Fig. 9.15(c) shows a turbo decoder consisting of two decoders concatenated and separated by an interleaver where one decoder processes the systematic (message) bit sequence y^s and the parity bit sequence y^{1p}/y^{2p} together with the extrinsic information L_{ej} (provided by the other decoder) to produce the information L_{ei} and provides it to the other decoder in an iterative manner. The turbo encoder and the demultiplexer are cast into the MATLAB routines 'encoder()' and 'demultiplex()', respectively. Now, let us see how the two types of decoder, each implementing the log-MAP (maximum a posteriori probability) algorithm and the SOVA (soft-out Viterbi algorithm), are cast into the MATLAB routines 'logmap()' and 'sova()', respectively.

<Log-MAP (Maximum a Posteriori Probability) Decoding cast into 'logmap()'>

To understand the operation of the turbo decoder, let us begin with the definition of *priori LLR* (*log-likelihood ratio*), called a priori L-value, which is a soft value measuring how high the probability of a binary random variable \mathbf{u} being +1 is in comparison with that of \mathbf{u} being -1:

$$L_{\mathbf{u}}(u) = \ln \frac{P_{\mathbf{u}}(u=+1)}{P_{\mathbf{u}}(u=-1)} \quad \text{with } P_{\mathbf{u}}(u): \text{ the probability of } \mathbf{u} \text{ being } u \quad (9.4.33)$$

This is a priori information known before the result \mathbf{y} caused by \mathbf{u} becomes available. While the sign of LLR

$$\hat{u} = \text{sign}\{L_{\mathbf{u}}(u)\} = \begin{cases} +1 & P_{\mathbf{u}}(u=+1) > P_{\mathbf{u}}(u=-1) \\ -1 & P_{\mathbf{u}}(u=+1) < P_{\mathbf{u}}(u=-1) \end{cases} \quad (9.4.34)$$

is a hard value denoting whether or not the probability of \mathbf{u} being +1 is higher than that of \mathbf{u} being -1, the magnitude of LLR is a soft value describing the reliability of the decision based on \hat{u} . Conversely, $P_{\mathbf{u}}(u=+1)$ and $P_{\mathbf{u}}(u=-1)$ can be derived from $L_{\mathbf{u}}(u)$:

$$P_{\mathbf{u}}(u=+1) \stackrel{(9.4.33)}{=} e^{L(u)} P_{\mathbf{u}}(u=-1) \xrightarrow{P(u=+1)+P(u=-1)=1} P_{\mathbf{u}}(u=+1) = \frac{e^{L(u)}}{1+e^{L(u)}} \text{ and } P_{\mathbf{u}}(u=-1) = \frac{1}{1+e^{L(u)}}$$

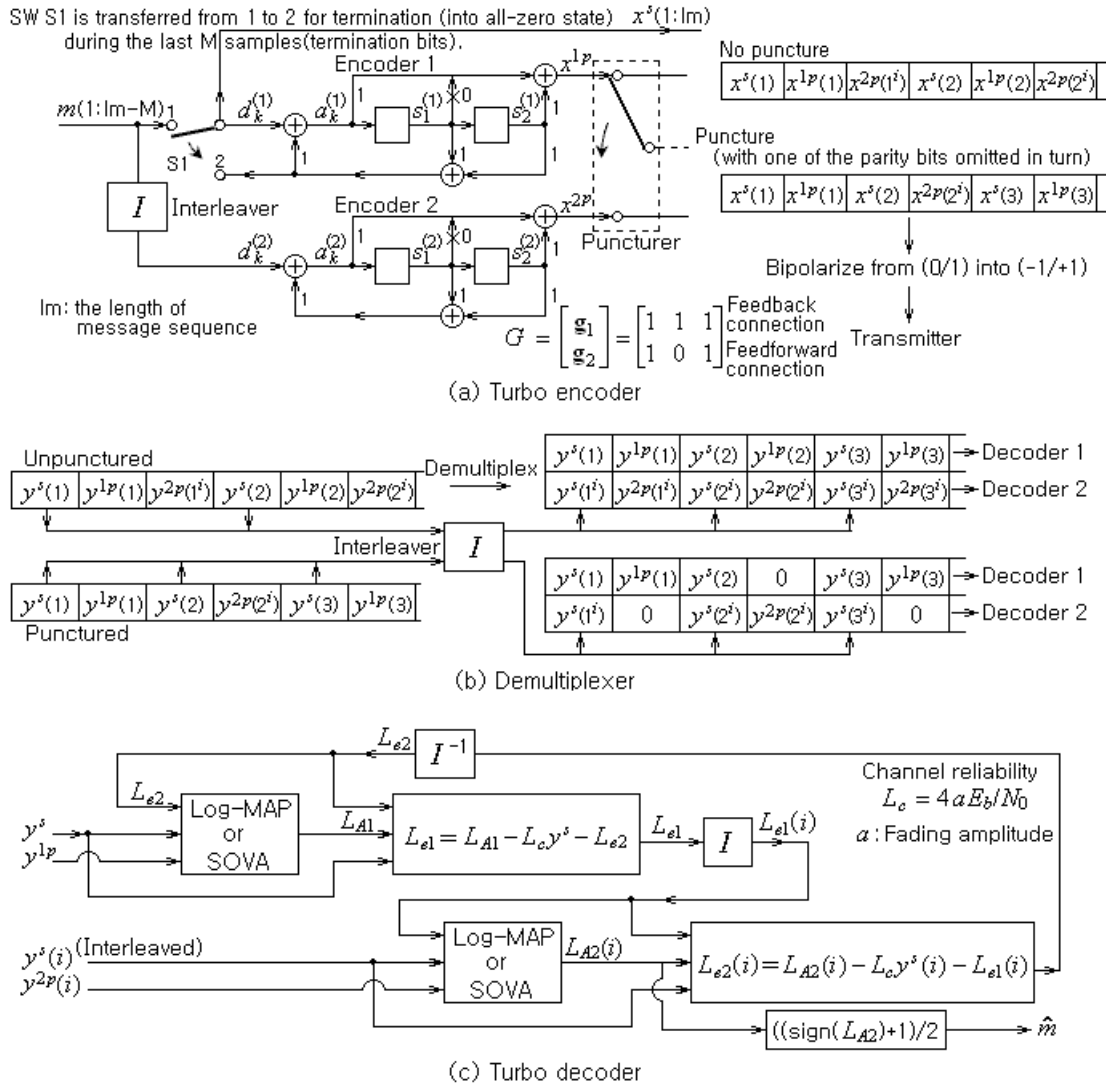


Figure 9.15 A typical turbo coding

```

function y = demultiplex(r,map,puncture)
%Copyright 1998, Yufei Wu, MPRG lab, Virginia Tech. for academic use
% map: Interleaver mapping
Nb = 3-puncture; lu = length(r)/Nb;
if puncture==0 % unpunctured
    for i=1:lu, y(:,2*i) = r(3*i-[1 0]).'; end
else % punctured
    for i=1:lu
        i2 = i*2;
        if rem(i,2)>0, y(:,i2)=[r(i2); 0]; else y(:,i2)=[0; r(i2)]; end
    end
end
sys_bit_seq = r(1,1:Nb:end); % the systematic bits for both decoders
y(:,1:2:lu*2) = [sys_bit_seq; sys_bit_seq(map)];

```

```

function x = rsc_encode(G,m,termination)
% Copyright 1998, Yufei Wu, MPRG lab, Virginia Tech. for academic use
% encodes a binary data block m (0/1) with a RSC (recursive systematic
% convolutional) code defined by generator matrix G, returns the output
% in x (0/1), terminates the trellis with all-0 state if termination>0
if nargin<3, termination = 0; end
[N,L] = size(G); % Number of output bits, Constraint length
M = L-1; % Dimension of the state
lu = length(m)+(termination>0)*M; % Length of the input
lm = lu-M; % Length of the message
state = zeros(1,M); % initialize the state vector
% To generate the codeword
x = [];
for i = 1:lu
    if termination<=0 | (termination>0 & i<=L_info)
        d_k = m(i);
    elseif termination>0 & i>lm
        d_k = rem(G(1,2:L)*state.',2);
    end
    a_k = rem(G(1,:)*[d_k state].',2);
    xp = rem(G(2,:)*[a_k state].',2); % 2nd output (parity) bits
    state = [a_k state(1:M-1)]; % Next state
    x = [x [d_k; xp]]; % since systematic, first output is input bit
end

```

```

function x = encoderm(m,G,map,puncture)
% Copyright 1998, Yufei Wu, MPRG lab, Virginia Tech. for academic use
% map: Interleaver mapping
% If puncture=0 (unpunctured), it operates with a code rate of 1/3.
% If puncture>0 (punctured), it operates with a code rate of 1/2.
% Multiplexer chooses odd/even-numbered parity bits from RSC1/RSC2.
[N,L] = size(G); % Number of output pits, Constraint length
M = L-1; % Dimension of the state
lm = length(m); % Length of the information message block
lu = lm + M; % Length of the input sequence
% 1st RSC coder output
x1 = rsc_encode(G,m,1);
% interleave input to second encoder
mi = x1(1,map); x2 = rsc_encode(G,mi,0);
% parallel to serial multiplex to get the output vector
x = [];
if puncture==0 % unpunctured, rate = 1/3;
    for i=1:lu
        x = [x x1(1,i) x1(2,i) x2(2,i)];
    end
else % punctured into rate 1/2
    for i=1:lu
        if rem(i,2), x = [x x1(1,i) x1(2,i)]; % odd parity bits from RSC1
        else x = [x x1(1,i) x2(2,i)]; % even parity bits from RSC2
        end
    end
end
end
x = 2*x - 1; % into bipolar format (+1/-1)

```

This can be expressed as

$$P_{\mathbf{u}}(u) = \frac{e^{(u+1)L(u)/2}}{1+e^{L(u)}} = \begin{cases} e^{L(u)/(1+e^{L(u)})} & \text{for } u=+1 \\ 1/(1+e^{L(u)}) & \text{for } u=-1 \end{cases} \quad (9.4.35)$$

Also, we define the *conditioned LLR*, which is used to detect the value of \mathbf{u} based on the value of another random variable \mathbf{y} affected by \mathbf{u} , as the *LAPP (Log A Posteriori Probability)*:

$$L_{\mathbf{u}|\mathbf{y}}(u | y) = \ln \frac{P_{\mathbf{u}}(u=+1|\mathbf{y})}{P_{\mathbf{u}}(u=-1|\mathbf{y})} \stackrel{(2.1.4)}{=} \ln \frac{P(y|u=+1)P_{\mathbf{u}}(u=+1)/P(y)}{P(y|u=-1)P_{\mathbf{u}}(u=-1)/P(y)} = \ln \frac{P(y|u=+1)}{P(y|u=-1)} + \ln \frac{P_{\mathbf{u}}(u=+1)}{P_{\mathbf{u}}(u=-1)} \quad (9.4.36)$$

Now, let \mathbf{y} be the output of a fading AWGN (additive white Gaussian noise) channel (with fading amplitude a and SNR per bit E_b/N_0) given \mathbf{u} as the input. Then, this equation for the conditioned LLR can be written as

$$L_{\mathbf{u}|\mathbf{y}}(u | y) = \ln \frac{\exp(-(E_b/N_0)(y-a)^2)}{\exp(-(E_b/N_0)(y+a)^2)} + \ln \frac{P_{\mathbf{u}}(u=+1)}{P_{\mathbf{u}}(u=-1)} = 4ay \frac{E_b}{N_0} + L_{\mathbf{u}}(u) = L_c y + L_{\mathbf{u}}(u) \quad (9.4.37)$$

$$\text{with } L_c = 4a \frac{E_b}{N_0} : \text{ the channel reliability}$$

The objective of *BCJR (Bahl-Cocke-Jelinek-Raviv) MAP (Maximum A posteriori Probability) algorithm* proposed in [B-1] is to detect the value of the k th message bit u_k depending on the sign of the following LAPP function:

$$\begin{aligned} L_A(u_k) &= \ln \frac{P_{\mathbf{u}}(u_k=+1|\mathbf{y})}{P_{\mathbf{u}}(u_k=-1|\mathbf{y})} = \ln \frac{\sum_{(s',s) \in S^+} p(s_k=s', s_{k+1}=s, \mathbf{y}) / p(\mathbf{y})}{\sum_{(s',s) \in S^-} p(s_k=s', s_{k+1}=s, \mathbf{y}) / p(\mathbf{y})} \\ &= \ln \frac{\sum_{(s',s) \in S^+} p(s_k=s', s_{k+1}=s, \mathbf{y})}{\sum_{(s',s) \in S^-} p(s_k=s', s_{k+1}=s, \mathbf{y})} \end{aligned} \quad (9.4.38)$$

with S^+ / S^- : the set of all the encoder state transitions from s' to s caused by $u_k = +1/-1$

Note that P and p denote the probability of a discrete-valued random variable and the probability density of a continuous-valued random variable. The numerator/denominator of this LAPP function are the sum of the probabilities that the channel output to $u_k = +1/-1$ will be $\mathbf{y} = \{\mathbf{y}_{j < k}, y_k, \mathbf{y}_{j > k}\}$ with the encoder state transition from s' to s where each joint probability density $p(s', s, \mathbf{y}) = p(s_k = s', s_{k+1} = s, \mathbf{y})$ can be written as

$$p(s', s, \mathbf{y}) = p(s_k = s', s_{k+1} = s, \mathbf{y}) = p(s', \mathbf{y}_{j < k}) p(s, y_k | s') p(\mathbf{y}_{j > k} | s) = \alpha_{k-1}(s') \gamma_k(s', s) \beta_k(s) \quad (9.4.39)$$

where $\alpha_{k-1}(s') = p(s', \mathbf{y}_{j < k})$ is the probability that the state s_k at the k th depth level (stage) in the trellis is s' with the output sequence $\mathbf{y}_{j < k}$ generated before the k th level, $\gamma_k(s', s) = p(s, y_k | s')$ is the probability that the state transition from $s_k = s'$ to $s_{k+1} = s$ is made with the output y_k generated, and $\beta_k(s) = p(\mathbf{y}_{j > k} | s)$ is the probability that the state s_{k+1} is s with the output sequence

$\mathbf{y}_{j>k}$ generated after the k th level. The first and third factors $\alpha_{k-1}(s')/\beta_k(s)$ can be computed in a forward/backward recursive way:

$$\begin{aligned}\alpha_k(s) &= p(s, \mathbf{y}_{j \leq k+1}) = p(s', \mathbf{y}_{j \leq k}, s, y_k) = \sum_{s' \in S} p(s', \mathbf{y}_{j \leq k}) p(s, y_k | s') \\ &= \sum_{s' \in S} \alpha_{k-1}(s') \gamma_k(s', s) \text{ with } \alpha_0(0) = 1, \alpha_0(s) = 0 \text{ for } s \neq 0\end{aligned}\quad (9.4.40)$$

$$\begin{aligned}\beta_{k-1}(s') &= p(\mathbf{y}_{j > k-1} | s') = p(s', y_k, s, \mathbf{y}_{j > k} | s') = \sum_{s \in S} p(s, y_k | s') p(\mathbf{y}_{j > k} | s) \\ &= \sum_{s \in S} \gamma_k(s', s) \beta_k(s)\end{aligned}\quad (9.4.41a)$$

with

$$\beta_K(s) = \begin{cases} \beta_K(0)=1 \text{ and } \beta_K(s)=0 \forall s \neq 0 & \text{if terminated at all-zero state} \\ \beta_K(s)=1/N_s \forall s & \text{otherwise} \end{cases}\quad (9.4.41b)$$

where $N_s = 2^{L-1}$ (L : the constraint length) is the number of states and K is the number of decoder input symbols. The second factor $\gamma_k(s', s)$ can be found as

$$\begin{aligned}\gamma_k(s', s) &= p(s, y_k | s') \stackrel{(2.1.4)}{=} \frac{p(s', s, y_k)}{p(s')} = \frac{p(s', s) p(y_k | s', s)}{p(s')} \\ &= p(s_{k+1} | s'_k) p(y_k | s'_k, s_{k+1}) = p(u_k) p(y_k | u_k) = p(u_k) p(y_k^s, y_k^p | u_k, x_k^p(u_k)) \\ &\stackrel{(9.4.35)}{=} \frac{e^{(u+1)L(u)/2}}{1+e^{L(u)}} \exp\left(-\frac{E_b}{N_0}(y_k^s - a u_k)^2 - \frac{E_b}{N_0}(y_k^p - a x_k^p(u_k))^2\right) \\ &\quad \text{AWGN channel with fading amplitude } a \text{ and SNR per bit } E_b/N_0 \\ &= \frac{e^{(u+1)L(u)/2}}{1+e^{L(u)}} A_k \exp\left(2a \frac{E_b}{N_0}(y_k^s u_k + y_k^p x_k^p(u_k))\right) \text{ where } u_k^2 = 1 \\ &= \frac{e^{(u+1)L(u)/2}}{1+e^{L(u)}} A_k \exp\left(\frac{1}{2} L_c [y_k^s \ y_k^p] \begin{bmatrix} u_k \\ x_k^p(u_k) \end{bmatrix}\right)\end{aligned}\quad (9.4.42)$$

$$\text{with } A_k = \exp\left(-\frac{E_b}{N_0}((y_k^s)^2 + a^2 u_k^2 + (y_k^p)^2 + a^2 (x_k^p(u_k))^2)\right) \text{ and } L_c = 4a \frac{E_b}{N_0} \text{ (channel reliability)}$$

Note a couple of things about this equation:

- To compute γ_k , we need to know the channel fading amplitude a and the SNR per bit E_b/N_0 .
- A_k does not have to be computed since it will be substituted directly or via α_k (Eq. (9.4.40)) or β_k (Eq. (9.4.41)) into Eq. (9.4.39), then substituted into both the numerator and the denominator of Eq. (9.4.38), and finally cancelled.

The following MATLAB routine 'logmap()' corresponding to the block named 'Log-MAP or SOVA' in Fig. 9.15(c) uses these equations to compute the LAPP function (9.4.38). Note that in the routine, the multiplications of the exponential terms are done by adding their exponents and that is why Alpha and Beta (each representing the exponent of α_k and β_k) are initialized to a large negative number as $-\text{Infy} = -100$ (corresponding to a nearly-zero $e^{-100} \approx 0$) under the assumption of initial all-zero state and for the termination of decoder 1 in all-zero state, respectively. (Q: Why is Beta initialized to $-\ln N_s$ ($-\log(N_s)$) for non-termination of decoder 2?)

```

function L_A = logmap(Ly,G,Lu,ind_dec)
% Copyright 1998, Yufei Wu, MPRG lab, Virginia Tech. for academic use
% Log_MAP algorithm using straightforward method to compute branch cost
% Input: Ly      = scaled received bits Ly=0.5*L_c*y=(2*a*rate*Eb/N0)*y
%           G      = code generator for the RSC code a in binary matrix
%           Lu      = extrinsic information from the previous decoder.
%           ind_dec= index of decoder=1/2 (assumed to be terminated/open)
% Output: L_A     = ln (P(x=1|y)/P(x=-1|y)), i.e., Log-Likelihood Ratio
%                (soft-value) of estimated message input bit at each level
lu=length(Ly)/2; Infy=1e2; EPS=1e-50; % Number of input bits, etc
[N,L] = size(G);
Ns = 2^(L-1); % Number of states in the trellis
Le1=-log(1+exp(Lu)); Le2=Lu+Le1; % ln(exp((u+1)/2*Lu)/(1+exp(Lu)))
% Set up the trellis
[nout, ns, pout, ps] = trellis(G);
% Initialization of Alpha and Beta
Alpha(1,2:Ns) = -Infy; % Eq.(9.4.40) (the initial all-zero state)
if ind_dec==1 % for decoder D1 with termination in all-zero state
    Beta(lu+1,2:Ns) = -Infy; % Eq.(9.4.41b) (the final all-zero state)
else % for decoder D2 without termination
    Beta(lu+1,:) = -log(Ns)*ones(1,Ns);
end
% Compute gamma at every depth level (stage)
for k = 2:lu+1
    Lyk = Ly(k*2-[3 2]); gam(:, :, k) = -Infy*ones(Ns,Ns);
    for s2 = 1:Ns % Eq.(9.4.42)
        gam(ps(s2,1),s2,k) = Lyk*[-1 pout(s2,2)].' +Le1(k-1);
        gam(ps(s2,2),s2,k) = Lyk*[+1 pout(s2,4)].' +Le2(k-1);
    end
end
% Compute Alpha in forward recursion
for k = 2:lu
    for s2 = 1:Ns
        alpha = sum(exp(gam(:,s2,k).'+Alpha(k-1,:))); % Eq.(9.4.40)
        if alpha<EPS, Alpha(k,s2)=-Infy; else Alpha(k,s2)=log(alpha); end
    end
    tempmax(k) = max(Alpha(k,:)); Alpha(k,:) = Alpha(k,:)-tempmax(k);
end
% Compute Beta in backward recursion
for k = lu:-1:2
    for s1 = 1:Ns
        beta = sum(exp(gam(s1,:,k+1)+Beta(k+1,:))); % Eq.(9.4.41)
        if beta<EPS, Beta(k,s1)=-Infy; else Beta(k,s1)=log(beta); end
    end
    Beta(k,:) = Beta(k,:) - tempmax(k);
end
% Compute the soft output LLR for the estimated message input
for k = 1:lu
    for s2 = 1:Ns % Eq.(9.4.39)
        temp1(s2)=exp(gam(ps(s2,1),s2,k+1)+Alpha(k,ps(s2,1))+Beta(k+1,s2));
        temp2(s2)=exp(gam(ps(s2,2),s2,k+1)+Alpha(k,ps(s2,2))+Beta(k+1,s2));
    end
    L_A(k) = log(sum(temp2)+EPS) - log(sum(temp1)+EPS); % Eq.(9.4.38)
end

```



```

function [nout,nstate,pout,pstate] = trellis(G)
% copyright 1998, Yufei Wu, MPRG lab, Virginia Tech for academic use
% set up the trellis with code generator G in binary matrix form.
% G: Generator matrix with feedback/feedforward connection in row 1/2
% e.g. G=[1 1 1; 1 0 1] for the turbo encoder in Fig. 9.15(a)
% nout(i,1:2): Next output [xs=m xp] (-1/+1) for state=i, message in=0
% nout(i,3:4): next output [xs=m xp] (-1/+1) for state=i, message in=1
% nstate(i,1): next state(1,...,2^M) for state=i, message input=0
% nstate(i,2): next state(1,...,2^M) for state=i, message input=1
% pout(i,1:2): previous out [xs=m xp] (-1/+1) for state=i, message in=0
% pout(i,3:4): previous out [xs=m xp] (-1/+1) for state=i, message in=1
% pstate(i,1): previous state having come to state i with message in=0
% pstate(i,2): previous state having come to state i with message in=1
% See Fig. 9.16 for the meanings of the output arguments.
[N,L] = size(G); % Number of output bits and Constraint length
M=L-1; Ns=2^M; % Number of bits per state and Number of states
% Set up next_out and next_state matrices for RSC code generator G
for state_i=1:Ns
    state_b = deci2bin1(state_i-1,M); % Binary state
    for input_bit=0:1
        d_k = input_bit;
        a_k=rem(G(1,:)*[d_k state_b]',2); % Feedback in Fig.9.15(a)
        out(input_bit+1,:)= [d_k rem(G(2,:)*[a_k state_b]',2)]; % Forward
        state(input_bit+1,:)= [a_k state_b(1:M-1)]; % Shift register
    end
    nout(state_i,:) = 2*[out(1,:) out(2,:)]-1; % bipolarize
    nstate(state_i,:) = [bin2deci(state(1,:)) bin2deci(state(2,:))]+1;
end
% Possible previous states having reached the present state
% with input_bit=0/1
for input_bit=0:1
    bN = input_bit*N; b1 = input_bit+1; % Number of output bits = 2;
    for state_i=1:Ns
        pstate(nstate(state_i,b1),b1) = state_i;
        pout(nstate(state_i,b1),bN+[1:N]) = nout(state_i,bN+[1:N]);
    end
end
end

```

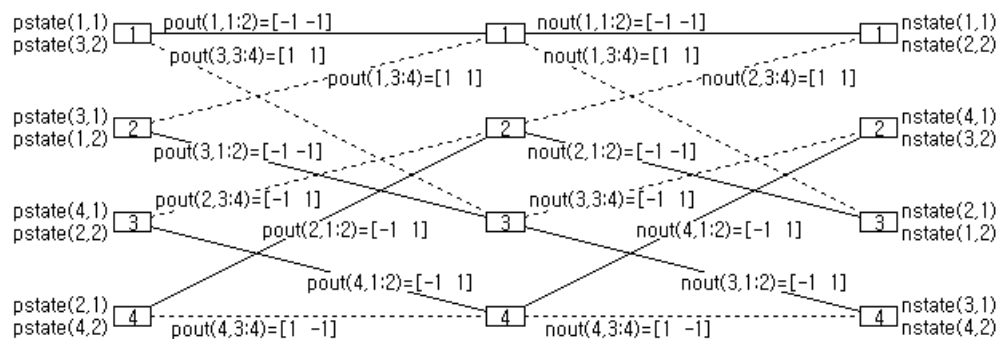


Figure 9.16 A trellis built by the routine 'trellis()' and its output arguments

<SOVA (Soft-In/Soft-Output Viterbi Algorithm) Decoding cast into ‘sova()’> [H-2]

The objective of the SOVA-MAP decoding algorithm is to find the state sequence $\mathbf{s}^{(i)}$ and the corresponding input sequence $\mathbf{u}^{(i)}$ which maximizes the following MAP (maximum a posteriori probability) function:

$$P(\mathbf{s}^{(i)} | \mathbf{y}) \stackrel{(2.1.4)}{=} p(\mathbf{y} | \mathbf{s}^{(i)}) \frac{P(\mathbf{s}^{(i)})}{p(\mathbf{y})} \propto p(\mathbf{y} | \mathbf{s}^{(i)}) P(\mathbf{s}^{(i)}) \text{ for given } \mathbf{y} \quad (9.4.43)$$

This probability would be found from the multiplications of the branch transition probabilities defined by Eq. (9.4.42). However, as is done in the routine ‘logmap()’, we will compute the path metric by accumulating the logarithm or exponent of only the terms affected by $u_k^{(i)}$ as follows:

$$M_k(\mathbf{s}^{(i)}) = M_{k-1}(\mathbf{s}^{(i)}) + \frac{L(u)}{2} u_k^{(i)} + \frac{1}{2} L_c[y_k^s \ y_k^p] \begin{bmatrix} u_k^{(i)} \\ x_k^p(u_k^{(i)}) \end{bmatrix} \quad (9.4.44)$$

The decoding algorithm cast into the routine ‘sova(Ly,G,Lu,ind_dec)’ proceeds as follows:

- (Step 0) Find the number of the $[y_k^s \ y_k^p]$ ’s in Ly given as the first input argument: $l_u = \text{length}(\text{Ly})/2$. Find the number N of output bits of the two encoders and the constraint length L from the row and column dimensions of the generator matrix G . Let the number of states be $N_s = 2^{L-1}$, the SOVA window size $\delta=30$, and the depth level $k=0$. Under the assumption of all-zero state at the initial stage (depth level zero), initialize the path metric to $M_k(s_0) = 0 = \ln 1$ (corresponding to probability 1) only for the all-zero state s_0 and to $M_k(s_j) = -\infty = \ln 0$ (corresponding to probability 0) for the other states s_j ($j \neq 0$).
- (Step 1) Increment k by one and determine which one of the hypothetical encoder input (message) $u_{k-1}=0$ or $u_{k-1}=1$ would result in larger path metric $M_k(s_i)$ (computed by Eq. (9.4.44)) for every state s_i ($i=0:N_s-1$) at level k and chooses the corresponding path as the survivor path, storing the estimated value of u_{k-1} (into ‘pinput(i,k)’) and the relative path metric difference ‘DM(i,k)’ of the survivor path over the other (non-surviving) path

$$\Delta M_k(s_i) = M_k(s_i | u_{k-1} = 0/1) - M_k(s_i | u_{k-1} = 1/0) \quad (9.4.45)$$

for every state at the stage. Repeat this step (in the forward direction) till $k=l_u$.

- (Step 2) Depending on the value of the fourth input argument ‘ind_dec’, determine the all-zero state s_0 or any state belonging to the most likely path (with $\text{Max } M_k(s_i)$) to be the final state $\hat{s}(k)$ (sh(k)).
- (Step 3) Find $\hat{u}(k)$ (uhat(k)) from ‘pinput(i,k)’ (constructed at Step 1) and the corresponding previous state $\hat{s}(k-1)$ (shat(k-1)) from the trellis structure. Decrement k by one. Repeat this step (in the backward direction) till $k=0$.
- (Step 4) To find the reliability of $\hat{u}(k)$, let $\text{LLR} = \Delta M_k(\hat{s}(k))$. Trace back the non-surviving paths from the optimal states $\hat{s}(k+i)$ (for $i=1:\delta$ such that $k+i \leq l_u$), find the nearly-optimal input $\hat{u}_i(k)$. If $\hat{u}_i(k) \neq \hat{u}(k)$ for some i , let $\text{LLR} = \text{Min}\{\text{LLR}, \Delta M_{k+i}(\hat{s}(k+i))\}$. In this way, find the LLR estimate and multiply it with the bipolarized value of $\hat{u}(k)$ to determine the soft output or L-value:

$$L_A(\hat{u}(k)) = (2\hat{u}(k) - 1) \text{LLR} \quad (9.4.46)$$

```

function L_A = sova(Ly,G,Lu,ind_dec)
% Copyright: Yufei Wu, 1998, MPRG lab, Virginia Tech for academic use
% This implements Soft Output Viterbi Algorithm in trace back mode
% Input: Ly      : Scaled received bits Ly=0.5*L_c*y=(2*a*rate*Eb/N0)*y
%           G      : Code generator for the RSC code in binary matrix form
%           Lu      : Extrinsic information from the previous decoder.
%           ind_dec: Index of decoder=1/2
%                   (assumed to be terminated in all-zero state/open)
% Output: L_A     : Log-Likelihood Ratio (soft-value) of
%                   estimated message input bit u(k) at each stage,
%                   ln (P(u(k)=1|y)/P(u(k)=-1|y))
lu = length(Ly)/2; % Number of y=[ys yp] in Ly
lu1 = lu+1; Infty = 1e2;
[N,L] = size(G); Ns = 2^(L-1); % Number of states
delta = 30; % SOVA window size
% Make decision after 'delta' delay. Tracing back from (k+delta) to k,
% decide bit k when received bits for bit (k+delta) are processed.
% Set up the trellis defined by G.
[nout,ns,pout,ps] = trellis(G);
% Initialize the path metrics to -Infty
Mk(1:Ns,1:lu1)=-Infty; Mk(1,1)=0; % Only initial all-0 state possible
% Trace forward to compute all the path metrics
for k=1:lu
    Lyk = Ly(k*2-[1 0]); k1=k+1;
    for s=1:Ns % Eq. (9.4.44), Eq. (9.4.45)
        Mk0 = Lyk*pout(s,1:2).' -Lu(k)/2 +Mk(ps(s,1),k);
        Mk1 = Lyk*pout(s,3:4).' +Lu(k)/2 +Mk(ps(s,2),k);
        if Mk0>Mk1, Mk(s,k1)=Mk0; DM(s,k1)=Mk0-Mk1; pinput(s,k1)=0;
        else      Mk(s,k1)=Mk1; DM(s,k1)=Mk1-Mk0; pinput(s,k1)=1;
        end
    end
end
% Trace back from all-zero state or the most likely state for D1/D2
% to get input estimates uhat(k), and the most likely path (state) shat
if ind_dec==1, shat(lu1)=1; else [Max,shat(lu1)]=max(Mk(:,lu1)); end
for k=lu:-1:1
    uhat(k)=pinput(shat(k+1),k+1); shat(k)=ps(shat(k+1),uhat(k)+1);
end
% As the soft-output, find the minimum DM over a competing path
% with different information bit estimate.
for k=1:lu
    LLR = min(Infty,DM(shat(k+1),k+1));
    for i=1:delta
        if k+i<lu1
            u_ = 1-uhat(k+i); % the information bit
            tmp_state = ps(shat(k+i+1),u_+1);
            for j=i-1:-1:0
                pu=pinput(tmp_state,k+j+1); tmp_state=ps(tmp_state,pu+1);
            end
            if pu~=uhat(k), LLR = min(LLR,DM(shat(k+i+1),k+i+1)); end
        end
    end
    L_A(k) = (2*uhat(k)-1)*LLR; % Eq. (9.4.46)
end

```

```

%turbo_code_demo.m
% simulates the classical turbo encoding-decoding system.
% 1st encoder is terminated with tails bits. (lm+M) bits are scrambled
% and passed to 2nd encoder, which is left open without termination.
clear
dec_alg = 1; % 0/1 for Log-MAP/SOVA
puncture = 1; % puncture or not
rate = 1/(3-puncture); % Code rate
lu = 1000; % Frame size
Nframes = 100; % Number of frames
Niter = 4; % Number of iterations
EbN0dBs = 2.6; %[1 2 3];
N_EbN0dBs = length(EbN0dBs);
G = [1 1 1; 1 0 1]; % Code generator
a = 1; % Fading amplitude; a=1 in AWGN channel
[N,L]=size(G); M=L-1; lm=lu-M; % Length of message bit sequence
for nENDB = 1:N_EbN0dBs
    EbN0 = 10^(EbN0dBs(nENDB)/10); % convert Eb/N0[dB] to normal number
    L_c = 4*a*EbN0*rate; % reliability value of the channel
    sigma = 1/sqrt(2*rate*EbN0); % standard deviation of AWGN noise
    noes(nENDB,:) = zeros(1,Niter);
    for nframe = 1:Nframes
        m = round(rand(1,lm)); % information message bits
        [temp,map] = sort(rand(1,lu)); % random interleaver mapping
        x = encoderm(m,G,map,puncture); % encoder output [x(+1/-1)
        noise = sigma*randn(1,lu*(3-puncture));
        r = a.*x + noise; % received bits
        y = demultiplex(r,map,puncture); % input for decoder 1 and 2
        Ly = 0.5*L_c*y; % Scale the received bits
        for iter = 1:Niter
            % Decoder 1
            if iter<2, Lu1=zeros(1,lu); % Initialize extrinsic information
            else Lu1(map)=L_e2; % (deinterleaved) a priori information
            end
            if dec_alg==0, L_A1=logmap(Ly(1,:),G,Lu1,1); % all information
            else L_A1=sova(Ly(1,:),G,Lu1,1); % all information
            end
            L_e1= L_A1-2*Ly(1,1:2:2*lu)-Lu1; % Eq.(9.4.47)
            % Decoder 2
            Lu2 = L_e1(map); % (interleaved) a priori information
            if dec_alg==0, L_A2=logmap(Ly(2,:),G,Lu2,2); % all information
            else L_A2=sova(Ly(2,:),G,Lu2,2); % all information
            end
            L_e2= L_A2-2*Ly(2,1:2:2*lu)-Lu2; % Eq.(9.4.47)
            mhat(map)=(sign(L_A2)+1)/2; % Estimate the message bits
            noe(iter)=sum(mhat(1:lu-M)~=m); % Number of bit errors
        end % End of iter loop
        % Total number of bit errors for all iterations
        noes(nENDB,:) = noes(nENDB,:) + noe;
        ber(nENDB,:) = noes(nENDB,:)/nframe/(lu-M); % Bit error rate
        fprintf('\n');
    end % End of nframe loop
end % End of nENDB loop

```

Now, it is time to take a look at the main program “turbo_code_demo.m”, which uses the routine ‘logmap()’ or ‘sova()’ (corresponding to the block named ‘Log-MAP or SOVA’ in Fig. 9.15(c)) as well as the routines ‘encoderm()’ (corresponding to Fig. 9.15(a)), ‘rsc_encode()’, ‘demultiplex()’ (corresponding to Fig. 9.15(b)), and ‘trellis()’ to simulate the turbo coding system depicted in Fig. 9.15. All of the programs listed here in connection with turbo coding stem from the routines developed by Yufei Wu in the MPRG (Mobile/Portable Radio Research Group) of Virginia Tech. (Polytechnic Institute and State University). The following should be noted:

- One thing to note is that the *extrinsic information* L_e to be presented to one decoder i by the other decoder j should contain only the intrinsic information of decoder j that is obtained from its own parity bits not available to decoder i . Accordingly, one decoder should remove the information about y^s (available commonly to both decoders) and the priori information $L(u)$ (provided by the other decoder) from the overall information L_A to produce the information that will be presented to the other decoder. (Would your friend be glad if you gave his/her present back to him/her or presented him/her what he/she had already got?) To prepare an equation for this information processing job of each encoder, we extract only the terms affected by $u_k = \pm 1$ from Eqs. (9.4.44) and (9.4.42) (each providing the basis for the path metric (Eq. (9.4.45)) and LLR (Eq. (9.4.38)), respectively,) to write

$$\left(\frac{L(u)}{2} u_k^{(i)} + \frac{1}{2} L_c y_k^s u_k^{(i)} \right) \Big|_{u_k^{(i)}=+1} - \left(\frac{L(u)}{2} u_k^{(i)} + \frac{1}{2} L_c y_k^s u_k^{(i)} \right) \Big|_{u_k^{(i)}=-1} = L(u) + L_c y_k^s$$

which conforms with Eq. (9.4.37) for the conditioned LLR $L_{u|y}(u | y)$. To prepare the extrinsic information for the other decoder, this information should be removed from the overall information $L_A(u)$ produced by the routine ‘logmap()’ or ‘sova()’ as

$$L_e(u) = L_A(u) - L(u) - L_c y_k^s \quad (9.4.47)$$

- Another thing to note is that as shown in Fig. 9.15(c), the basis for the final decision about u is the deinterleaved overall information L_{A2} that is attributed to decoder 2. Accordingly, the turbo decoder should know the pseudo-random sequence ‘map’ (that has been used for interleaving by the transmitter) as well as the fading amplitude and SNR of the channel.
- The trellis structure and the output arguments produced by the routine ‘trellis()’ are illustrated in Fig. 9.16.

Interest readers are invited to run the program “turbo_code_demo.m” with the value of the control constant ‘dec_alg’ set to 0/1 for Log-MAP/SOVA decoding algorithm and see the BER becoming lower as the decoding iteration proceeds. How do the turbo codes work? How are the two decoding algorithms, Log-MAP and SOVA, compared? Is there any weakpoint of turbo codes? What is the measure against the weakpoint, if any? Unfortunately, to answer such questions is difficult for the authors and therefore, is beyond the scope of this book. As can be seen from the simulation results, turbo codes have an excellent BER performance close to the Shannon limit at low and medium SNRs. However, the decreasing rate of the BER curve of a turbo code can be very low at high SNR depending on the interleaver and the free distance of the code, which is called the ‘error floor’ phenomenon. Besides, turbo codes needs not only a large interleaver and block size but also many iterations to achieve such a good BER performance, which increases the complexity and latency (delay) of the decoder.

(d) Why don't we make the code for the above set of four symbols as follows?

Symbol	00	01	10	11
Codeword	0	1	10	11

With this code, we could have the average codeword length as

$$L = 0.81 \times 1 + 0.09 \times 1 + 0.09 \times 2 + 0.01 \times 2 = 1.1 [\text{bits/symbol}]$$

which is shorter than that with the code obtained in (a) or (b). To answer this question, think about the decoding of a coded sequence $\{1011\}$ based on the above table.

9.4 Channel Capacity and Hamming Codes

Recall that the MATLAB function 'Hammgen(n)' can be used to generate the $n \times N$ parity-check matrix for an (N, K) Hamming code where $N = 2^n - 1$ and $K = N - n$. Suppose the message sequence is coded with the (7,4) or (15,11) Hamming code, BPSK(Binary Phase Keying)-modulated, and then transmitted through a BSC (Binary Symmetric Channel).

(a) Compute the crossover (channel bit transmission error) probability of BPSK signaling with each of these two codings for SNRdB=0-13 [dB] by

$$\varepsilon \stackrel{(7.3.5)}{=} Q\left(\sqrt{SNR_c R_c}\right) \text{ with } R_c (\text{Code rate}) = \frac{K}{N} \quad (\text{P9.4.1})$$

and substitute this into Eq. (9.3.13) to get the channel capacity

$$C_t \stackrel{(9.3.13)}{=} 1 + (1 - \varepsilon) \log_2(1 - \varepsilon) + \varepsilon \log_2 \varepsilon = 1 - H_b(\varepsilon) [\text{bits/symbol}] \quad (\text{P9.4.2})$$

Plot the channel capacity C_t vs. SNRdB=0-13 and use the graph to find the rough value of SNRdB at which the inequality (9.3.8) is marginally satisfied:

$$b_c = R_c b_{\text{BPSK}} \stackrel{b=1}{=} R_c \leq C_t [\text{bits/symbol}] \quad (\text{P9.4.3})$$

(b) Referring to the lower part of the MATLAB routine 'do_Hamming_code74()' (Sec. 9.4.2), use Eq. (9.4.11) to compute the theoretical (approximate) probabilities of message bit error for SNRdB=0-13 [dB] with no code, (7,4) Hamming code, and (15,11) Hamming code and plot them versus SNRdB. Use the theoretical BER curves to find the rough values of SNRdB at which the two codings have the same BER as that with no coding.

(c) Modify the MATLAB routine 'do_Hamming_code74()' so that it can simulate the BPSK signaling with (7,4) Hamming code and (15,11) Hamming code for SNRdB = 5, 7, 9, and 11 [dB]. Run it to plot the BERs versus SNRdB on the theoretical BER curves obtained in (b).

Table P9.4 BERs of BPSK signaling with (7,4) or (15,11) Hamming code

	SNRdB=5[dB]	SNRdB=7[dB]	SNRdB=9[dB]	SNRdB=11[dB]
No code	0.037679		0.002413	
(7,4) Hamming code		0.012422 (0.010996)		0.000118 (0.000080)
(15,11) Hamming code	0.060606 (0.038549)		0.001196 (0.000830)	

9.5 Effect of Coding on BER

To understand why the BER of simulation result is much higher than the theoretical value of Eq. (9.4.11) (Sec. 9.4.2), insert the following statements at the appropriate places in the MATLAB routine 'do_Hamming_code74()' (Sec. 9.4.2).

```
notbel= sum(r_sliced~=coded); notbec1= sum(r_c~=coded);
si= bin2deci(s); % syndrome
if notbel>0 & iter<30
    fprintf('\n # of transmitted bit errors %d -> ',notbel);
    fprintf('%d after correction (syndrome= %d)',notbec1,si);
end
```

Then run the routine with SNRdB=5 to answer the following questions:

- (i) Is a single bit error in a codeword always corrected?
- (ii) Is there any case where the number of bit errors is increased by a wrong correction?

What do you think is the reason for the big gap between the two BERs that are obtained from simulation and theoretical formula (9.4.11)?

9.6 BCH (Bose-Chaudhuri-Hocquenghem) Codes with Simulink

BCH codes constitute a powerful class of cyclic codes that provides a large selection of block lengths, code rates, alphabet sizes, and error-correcting capability as listed in Table 5.2 of [S-3]. According to the table, the $(N, K) = (15, 7)$ and $(31, 16)$ BCH codes are represented by the generator polynomial coefficient vectors

$$\mathbf{g}_1 = 721(\text{octal}) = [1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1] \quad (\text{P9.6.1})$$

$$\mathbf{g}_2 = 107657(\text{octal}) = [1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1] \quad (\text{P9.6.2})$$

and their error correcting capabilities are 2 and 3, respectively. Suppose the message sequence is coded with these two BCH codes, BPSK(binary phase-shift keying)-modulated, and then transmitted through a BSC (binary symmetric channel).

- (a) Let us try to make use of the MATLAB built-in function 'bchgenpoly()' to make the generator polynomial vectors for the $(31, 16)$ BCH code:

```
N=31; K=16;
gBCH=bchgenpoly(N,K); % Galois row vector in GF(2) representing
                        % the (N,K) BCH code
g=double(gBCH.x) % Extracting the elements from a Galois array
```

Does the resulting vector conform with \mathbf{g}_2 ?

- (b) It may take a long time to use the MATLAB built-in routine 'decode()' with such a big BCH code as \mathbf{g}_2 . If we prepare the syndrome table using 'E=syndtable(H)' beforehand and then use 'decode(coded,N,K,'cyclic',g,E)' with the prepared syndrome table E, it will save a lot of decoding time. Likewise, if we prepare the error pattern matrix E and the corresponding error pattern index vector epi using 'cyclic_decoder0' (Sec. 9.4.3) beforehand and then use 'cyclic_decoder(coded,N,K,g,E,epi)', it will save the decoding time, too. Complete the following MATLAB program "do_cyclic_codes.m" so that it can simulate a BPSK communication system with each of the above two BCH codes and plot the BERs for SNRdBs=[2 4 10] together with the theoretical BER curves for SNRdBs=[0:0.01:14] obtained from Eq. (9.4.11). Run the completed program to get the BER curves.

```

%do_cyclic_codes.m
clear, clf
gsymbols=['bo';'r+';'kx';'md';'g*'];
% Theoretical BER curves
SNRdBs=0:0.01:14; SNRs=10.^(SNRdBs/10); pemb_uncoded=Q(sqrt(SNRs));
semilogy(SNRdBs,pemb_uncoded,':'), hold on
use_decode = 0; % Use encode()/decode or not
MaxIter=1e5; Target_no_of_error=100; SNRdBs1=2:4:10;
for iter=1:2
    if iter==1, N=15; K=7; nceb=2; %(15,7) BCH code generator
    else      N=31; K=16; nceb=3; % (31,16) BCH code generator
    end
    gBCH=bchgenpoly(N,K); %Galois row vector representing (N,K) BCH code
    g=double(gBCH.x);
    clear('S')
    if use_decode>0, H=cyclgen(N,g); E=syndtable(?);
    else
        E= combis(N,nceb); % All error patterns
        for i=1:size(E,1)
            S(i,:) = cyclic_decoder0(E(i,:),N,K,g); % Syndrome
            epi(bin2deci(S(i,:))) = ?; % Error pattern indices
        end
    end
    end
    Rc = K/N; %code rate
    SNRcs=SNRs*Rc; sqrtSNRcs=sqrt(SNRcs);
    ets=Q(sqrt(SNRcs)); % transmitted bit error probability Eq.(7.3.5)
    pemb_t=prob_err_msg_bit(ets,N,nceb);
    semilogy(SNRdBs,pemb_t,gsymbols(iter,1),'Markersize',5)
    for iter1=1:length(SNRdBs1)
        SNRdB = SNRdBs1(iter1); SNR = 10.^(SNRdB/10);
        SNRc = SNR*Rc; sqrtSNRc = sqrt(SNRc);
        et=Q(sqrt(SNRc)); % transmitted bit error probability Eq.(7.3.5)
        K100 = K*100;
        nombe = 0;
        for iter2=1:MaxIter
            msg=randint(1,K100); % Message vector
            if use_decode==0, coded=cyclic_??????? (msg,N,K,g);
            else msg=reshape(msg,length(msg)/K,K);
                coded=?????? (msg,N,K,'cyclic',g);
            end
            r= 2*coded-1 + randn(size(coded))/sqrtSNRc; % Received vector
            r_sliced= 1*(r>0); % Sliced
            if use_decode==0
                decoded= cyclic_??????? (r_sliced,N,K,g,E,epi);
            else decoded= ?????? (r_sliced,N,K,'cyclic',g,E);
            end
            nombe= nombe + sum(sum(decoded~=msg));
            if nombe>Target_no_of_error, break; end
        end
        lm=iter2*K100; pemb=nombe/lm; % Message bit error probability
        semilogy(SNRdB,pemb,gsymbols(iter,:), 'Markersize',5)
    end
end
end

```

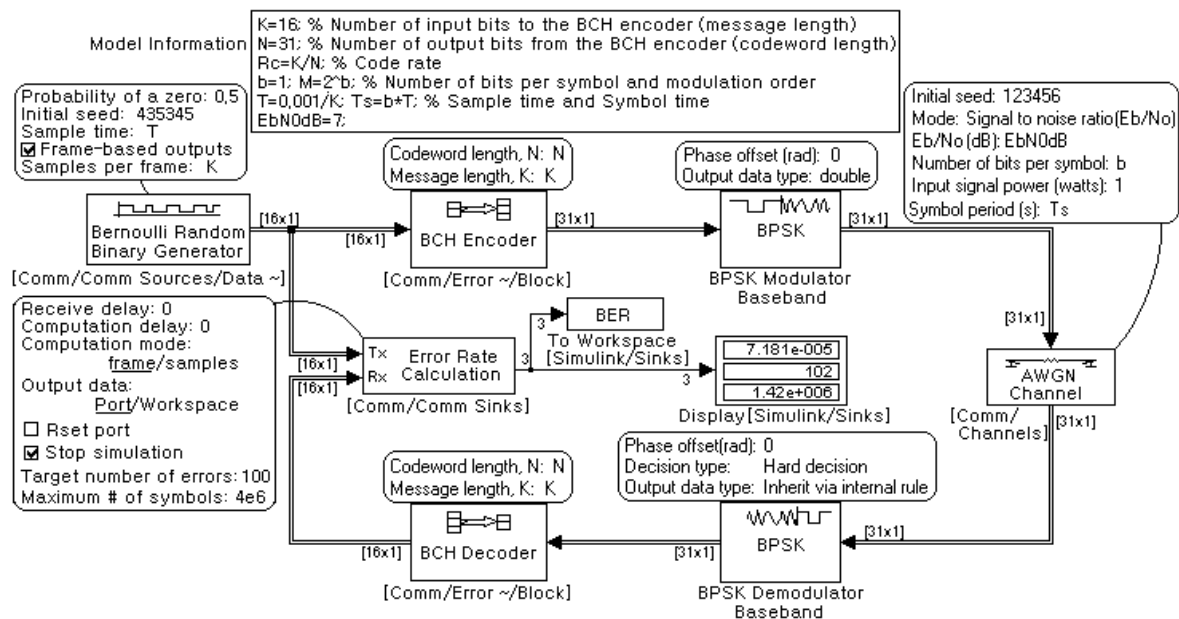



Figure P9.6 Simulink model "BCH_BPSK_sim.mdl"

- (c) Referring to the Simulink model "BCH_BPSK_sim.mdl" depicted in Fig. P9.6 and the following program "do_BCH_BPSK_sim.m", use Simulink to simulate a BPSK communication system with the (31,16) BCH code for error correction and find the BERs for $E_b/N_0 = -1, 3$, and 7dB. Noting that $SNR_{dB} = EbN0dB + 3$ since $SNR_b = E_b/(N_0/2)$, list the BERs together with those obtained using 'encode()/'decode()' in (b).

```
%do_BCH_BPSK_sim.m
clear, clf
K=16; % Number of input bits to the BCH encoder (message length)
N=31; % Number of output bits from the BCH encoder (codeword length)
Rc=K/N; % Code rate to be multiplied with the SNR in AWGN channel block
b=1; M=2^b; % Number of bits per symbol and modulation order
T=0.001/K; Ts=b*T; % Sample time and Symbol time
SNRbds=[2:4:10]; EbN0ds=SNRbds-3;
EbN0ds_t=0:0.1:10; EbN0s_t=10.^(EbN0ds_t/10);
SNRbds_t=EbN0ds_t+3;
BER_theory= prob_error(SNRbds_t, 'PSK', b, 'BER');
for i=1:length(EbN0ds)
    EbN0dB=EbN0ds(i);
    sim('BCH_BPSK_sim');
    BERs(i)=BER(1); % just ber among {ber, # of errors, total # of bits}
    fprintf(' With EbN0dB=%4.1f, BER=%10.4e=%d/%d\n', EbN0dB, BER);
end
semilogy(EbN0ds, BERs, 'r*', EbN0ds_t, BER_theory, 'b')
xlabel('Eb/N0 [dB]'); ylabel('BER');
title('BER of BCH code with BPSK');
```

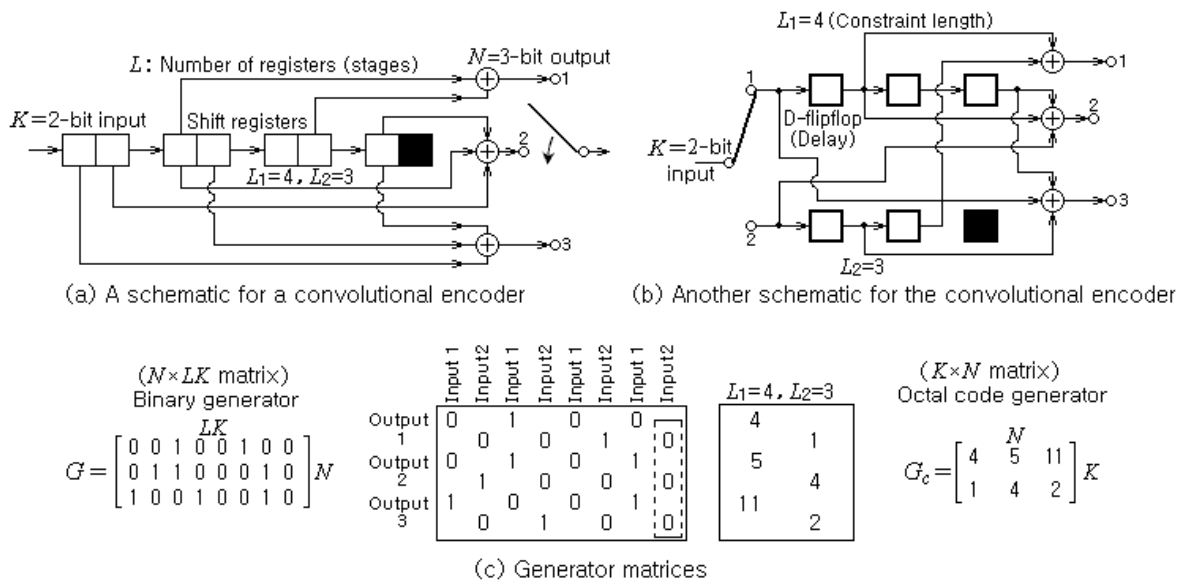


Figure P9.7 A convolutional encoder and its generator matrix

```
%dc09p07.m
% To practice using convenc() and vitdec() for channel coding
clear, clf
Gc=[4 5 11;1 4 2]; % Octal code generator matrix
K=size(Gc,1); % Number of encoder input bits
% Constraint length vector
Gc_m=max(Gc,');
for i=1:length(Gc_m), Lc(i)=length(dec2bin1(oct2dec(Gc_m(i))))); end
trel=poly2trellis(Lc,Gc);
Tbdepth=sum(Lc)*5; delay=Tbdepth*K;
lm=1e5; msg=randint(1,lm);
transmission_ber=0.02;
notbe=round(transmission_ber*lm); % Number of transmitted bit errors
ch_input=convenc([msg zeros(1,delay)],trel);
% Received/modulated/detected signal
ch_output= rem(randerr(1,length(ch_input),notbe)+ch_input,2);
decoded_trunc= vitdec(ch_output,trel,Tbdepth,'trunc','hard');
ber_trunc= sum(msg~=decoded_trunc(???))/lm;
decoded_cont= vitdec(ch_output,trel,Tbdepth,'cont','hard');
ber_cont=sum(msg~=decoded_cont(????????????))/lm;
% It is indispensable to use the delay for the decoding result
% obtained using vitdec(.,.,'cont',)
nn=[0:100-1];
subplot(221), stem(nn,msg(nn+1)), title('Message sequence')
subplot(223), stem(nn,decoded_cont(nn+1)), hold on
stem(delay,0,'rx')
decoded_term= vitdec(ch_output,trel,Tbdepth,'term','hard');
ber_term=sum(msg~=decoded_term(???))/lm;
fprintf('\n BER_trunc BER_cont BER_term')
fprintf('\n %9.2e %9.2e %9.2e\n', ber_trunc,ber_cont,ber_term)
```

9.7 A Convolutional Code and Viterbi Decoding

Fig. P9.7 shows a convolutional encoder (described by two kinds of schematic) and the two corresponding generator matrices where one is a binary generator matrix that is used by the 'conv_encoder()'/ 'conv_decoder()' and the other is an octal generator matrix that is used by the MATLAB built-in functions 'convenc()'/ 'vitdec()'. Let the convolutional encoder and the corresponding Viterbi decoder be used for channel coding where there happens a 2%-error in the 100,000 transmitted bits. Complete the above program "dc09p07.m" to simulate this situation and run it to find the BER.

9.8 A Convolutional Code and Viterbi Decoding with Simulink

Fig. P9.8.1 shows a convolutional encoder described by the octal code generator matrix $G_c = [133 \ 171]$. Fig. P9.8.2 shows the Simulink model "Viterbi_QAM_sim.mdl" that can be used to simulate a QAM communication with a given convolutional encoder and the corresponding Viterbi decoder. The following MATLAB function 'Viterbi_QAM()' can also be used to simulate a QAM communication with a given convolutional encoder and the corresponding Viterbi decoder. Complete the following MATLAB program "do_Viterbi_QAM", which uses the MATLAB function 'Viterbi_QAM()' and the Simulink model "Viterbi_QAM_sim.mdl" to simulate a 16-QAM communication system with the convolutional encoder of Fig. P9.8.1 and the corresponding Viterbi decoder. Run it to find the BERs for $E_b/N_0 = 3, 6$, and 9 dB.

	$E_b/N_0 = 3$ [dB]	$E_b/N_0 = 6$ [dB]	$E_b/N_0 = 9$ [dB]
MATLAB 'Viterbi_QAM.m'		0.006684	
Simulink 'Viterbi_QAM_sim.mdl'	0.466403		0.000079

```
function [pemb,nombe,notmb]=Viterbi_QAM(Gc,b,SNRbdB,MaxIter)
if nargin<4, MaxIter=1e5; end
if nargin<3, SNRbdB=5; end
if nargin<2, b=4; end
[K,N]=size(Gc); Rc=K/N; Gc_m=max(Gc,');
% Constraint length vector
for i=1:length(Gc_m), Lc(i)=length(dec2bin1(oct2dec(Gc_m(i)))); end
Nf=144; % Number of bits per frame
Nmod=Nf*N/K/b; % Number of QAM symbols per modulated frame
SNRb=10.^(SNRbdB/10); SNRbc=SNRb*Rc; sqrtSNRbc=sqrt(SNRbc);
sqrtSNRc=sqrt(2*b*SNRbc); % Complex noise for b-bit (coded) symbol
trell=poly2trellis(Lc,Gc);
Tbdepth=5; delay=Tbdepth*K;
nombe=0; Target_no_of_error=100;
for iter=1:MaxIter
    msg=randint(1,Nf); % Message vector
    coded= convenc(msg,trell); % Convolutional encoding
    modulated= QAM(coded,b); % 2^b-QAM-Modulation
    r= modulated + (randn(1,Nmod)+j*randn(1,Nmod))/sqrtSNRc;
    demodulated= QAM_dem(r,b); % 2^b-QAM-Demodulation
    decoded= vitdec(demodulated,trell,Tbdepth,'trunc','hard');
    nombe = nombe + sum(msg~=decoded(1:Nf)); %
    if nombe>Target_no_of_error, break; end
end
notmb=Nf*iter; % Number of total message bits
pemb=nombe/notmb; % Message bit error probability
```

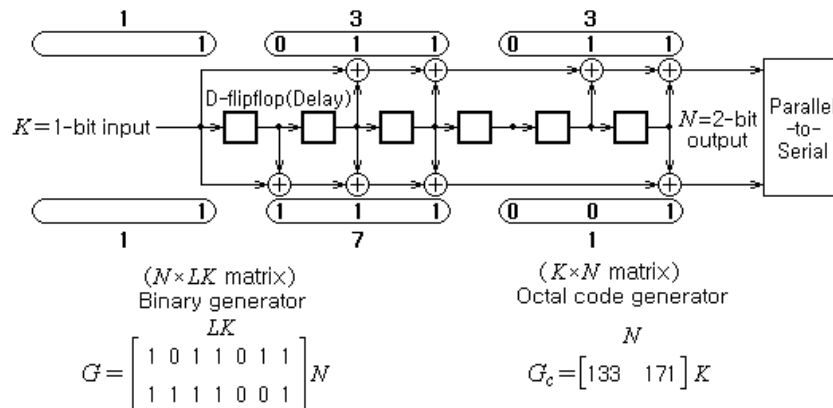


Figure P9.8.1 A convolutional encoder and its generator matrix

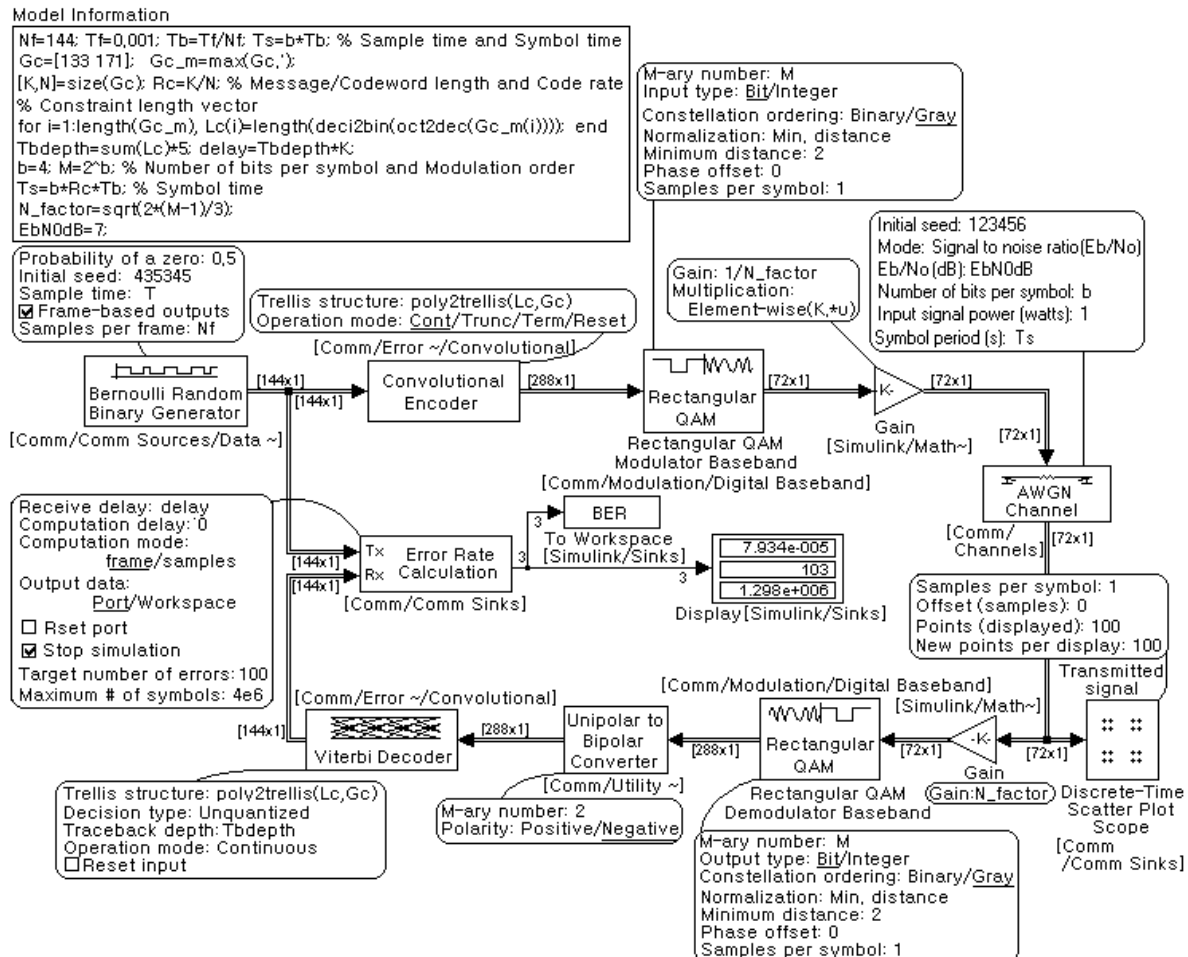


Figure P9.8.2 Simulink model "Viterbi_QAM_sim.mdl"

```

%do_Viterbi_QAM.m
clear, clf
Nf=144; Tf=0.001; Tb=Tf/Nf; % Frame size, Frame and Sample/Bit time
Gc=[133 171];
[K,N]=size(Gc); Rc=K/N; % Message/Codeword length and Code rate
% Constraint length vector
Gc_m=max(Gc. ');
for i=1:length(Gc_m)
    Lc(i)=length(deci2bin1(oct2dec(Gc_m(i))));
end
Tbdepth=sum(Lc)*5; delay=Tbdepth*K;
b=4; M=2^b; % Number of bits per symbol and Modulation order
Ts=b*Rc*Tb; % Symbol time
N_factor=sqrt(2*(M-1)/3); % Eq.(7.5.4a)
EbN0dBs_t=0:0.1:10; SNRbdBs_t=EbN0dBs_t+3;
BER_theory= prob_error(SNRbdBs_t, 'QAM', b, 'BER');
EbN0dBs=[3 6]; Target_no_of_error=50;
for i=1:length(EbN0dBs)
    EbN0dB=EbN0dBs(i); SNRbdB=EbN0dB+3;
    randn('state', 0);
    [pemb,nombe,notmb]=???????_QAM(Gc,b,SNRbdB,Target_no_of_error);
    pembs(i)=pemb;
    sim('Viterbi_QAM_sim'); pembs_sim(i)=BER(1);
end
[pembs; pembs_sim]
semilogy(EbN0dBs,pembs,'r*', EbN0dBs_t,BER_theory,'b')
xlabel('Eb/N0 [dB]'); ylabel('BER');

function qamseq=QAM(bitseq,b)
bpsym = nextpow2(max(bitseq)); % no of bits per symbol
if bpsym>0, bitseq = deci2bin(bitseq,bpsym); end
if b==1, qamseq=bitseq*2-1; return; end % BPSK modulation
% 2^b-QAM modulation
N0=length(bitseq); N=ceil(N0/b);
bitseq=bitseq(:).'; bitseq=[bitseq zeros(1,N*b-N0)];
b1=ceil(b/2); b2=b-b1; b21=b^2; b12=2^b1; b22=2^b2;
g_code1=2*gray_code(b1)-b12+1; g_code2=2*gray_code(b2)-b22+1;
tmp1=sum([1:2:2^b1-1].^2)*b21; tmp2=sum([1:2:b22-1].^2)*b12;
M=2^b; Kmod=sqrt(2*(M-1)/3);
%Kmod=sqrt((tmp1+tmp2)/2/(2^b/4)) % Normalization factor
qamseq=[];
for i=0:N-1
    bi=b*i; i_real=bin2deci(bitseq(bi+[1:b1]))+1;
    i_imag=bin2deci(bitseq(bi+[b1+1:b]))+1;
    qamseq=[qamseq (g_code1(i_real)+j*g_code2(i_imag))/Kmod];
end

function [g_code,b_code]=gray_code(b)
N=2^b; g_code=0:N-1;
if b>1, g_code=gray_code0(g_code); end
b_code=deci2bin(g_code);

function g_code=gray_code0(g_code)
N=length(g_code); N2=N/2;
if N>=4, N2=N/2; g_code(N2+1:N)=fftshift(g_code(N2+1:N)); end
if N>4, g_code=[gray_code0(g_code(1:N2))
gray_code0(g_code(N2+1:N))]; end

```

```

function bitseq=QAM_dem(qamseq,b,bpsym)
%BPSK demodulation
if b==1, bitseq=(qamseq>=0); return; end
%2^b-QAM demodulation
N=length(qamseq);
b1=ceil(b/2); b2=b-b1;
g_code1=2*gray_code(b1)-2^b1+1; g_code2=2*gray_code(b2)-2^b2+1;
tmp1=sum([1:2:2^b1-1].^2)*2^b2;
tmp2=sum([1:2:2^b2-1].^2)*2^b1;
Kmod=sqrt((tmp1+tmp2)/2/(2^b/4)); % Normalization factor
g_code1=g_code1/Kmod; g_code2=g_code2/Kmod;
bitseq=[];
for i=1:N
    [emin1,i1]=min(abs(real(qamseq(i))-g_code1));
    [emin2,i2]=min(abs(imag(qamseq(i))-g_code2));
    bitseq=[bitseq deci2bin1(i1-1,b1) deci2bin1(i2-1,b2)];
end
if (nargin>2)
    N = length(bitseq)/bpsym; bitmatrix = reshape(bitseq,bpsym,N).';
    for i=1:N, intseq(i)=bin2deci(bitmatrix(i,:)); end
    bitseq = intseq;
end

```

9.9 Ungerboeck TCM (Trellis-Coded Modulation) Codes

Figs. P9.9.1(a) and (b) show two Ungerboeck TCM encoders and Fig. P9.9.2 shows a Simulink model “TCM_sim.mdl” that simulates the TCM encoding-decoding with the TCM encoder of Fig. P9.9.1.

```

function [pemb,nombe,notmb]=...
    TCM(state_eq,K,Nsb,N,Constellation,SNRbdB,Target_no_of_error)
M=2^N; Rc=K/N;
Nf=288; % Number of bits per frame
Nmod=Nf/K; % Number of symbols per modulated frame
SNRb=10.^(SNRbdB/10); SNRbc=SNRb*Rc;
sqrtSNRc=sqrt(2*N*SNRbc); % Complex noise per K=Rc*N-bit symbol
nombe=0; MaxIter=1e6;
for iter=1:MaxIter
    msg=randint(1,Nf); % Message vector
    coded = TCM_encoder(state_eq,K,Nsb,N,msg,Constellation);
    r= coded + (randn(1,Nmod)+j*randn(1,Nmod))/sqrtSNRc;
    decoded= TCM_decoder(state_eq,K,Nsb,r,Constellation);
    nombe = nombe + sum(msg~=decoded(1:Nf));
    if nombe>Target_no_of_error, break; end
end
notmb=Nf*iter; % Number of total message bits
pemb=nombe/notmb; % Message bit error probability

function [s1,x]=TCM_state_eq1(s,u,Constellation)
% State equation for the TCM encoder in Fig. P9.9.1(a)
% Input: s= State, u= Input
% Output: s1= Next state, x= Output
s1 = [u(2) s(1)];
x= Constellation(bin2deci([u(1) rem(u(2)+s(2),2) s(1)]))+1);

```

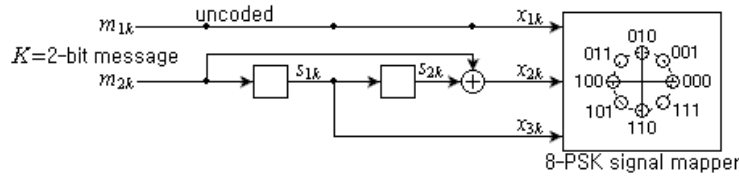
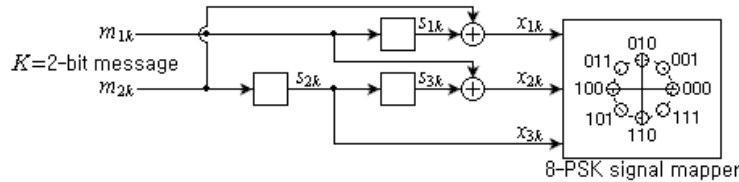
(a) A Ungerboeck TCM encoder consisting of 2^2 -state FSM and 8-PSK signal mapper(b) A Ungerboeck TCM encoder consisting of 2^3 -state FSM and 8-PSK signal mapper

Figure P9.9.1 Ungerboeck TCM encoder

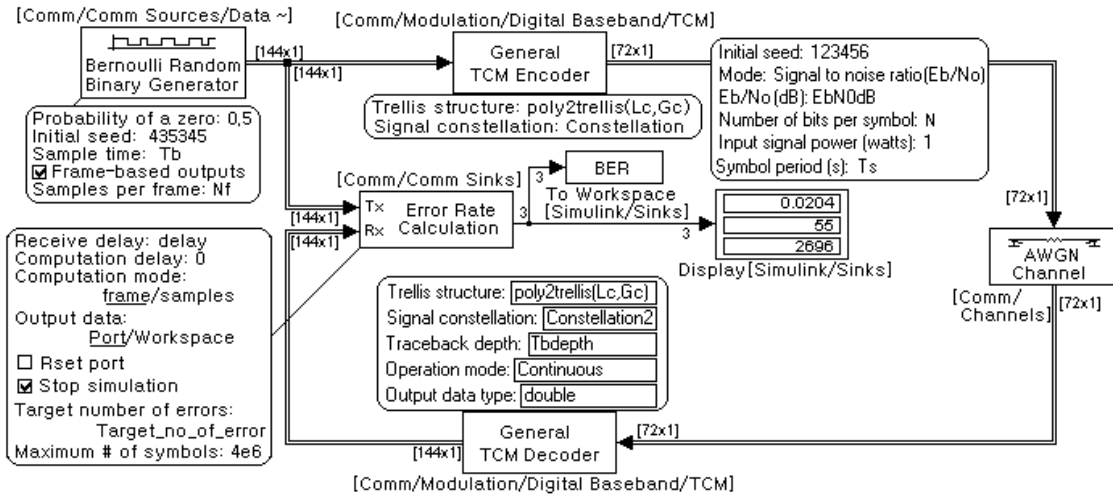


Figure P9.9.2 Simulink model for the Ungerboeck TCM of Figure P9.9.1 ("TCM_sim.mdl")

- (a) Run the following program "do_TCM_8PSK.m" to use the above MATLAB routine "TCM()" and the Simulink model "TCM_sim.mdl" for finding the BERs of the TCM system with the encoder-modulator of Fig. P9.9.1(a) where the SNR is given as $E_b/N_0 = 2$ [dB]. Set the 8PSK constellation for "TCM_sim.mdl" to C_1 or C_3 (Gray code).

$$\begin{aligned}
 C_1 &= \left\{ \exp\left(j \frac{2k\pi}{8}\right), k=0,1,2,3,4,5,6,7 \right\} \\
 C_2 &= \left\{ \exp\left(j \frac{2k\pi}{8}\right), k=0,4,2,6,1,5,3,7 \right\} \\
 C_3 &= \left\{ \exp\left(j \frac{2k\pi}{8}\right), k=0,1,3,2,6,7,5,4 \right\}
 \end{aligned} \tag{P9.9.1}$$

What difference will be made to the BER? Set the 8PSK constellation for "TCM.m" to C_2 or C_3 . What difference will be made to the BER? Fill in the following table.

Table P9.9 BER of TCM systems

E_b/N_0	BER			
	QPSK - no code (Theoretical)		TCM-8PSK (Fig. P9.9.1(a))	TCM-8PSK (Fig. P9.9.1(b))
2dB	0.038	TCM with C_1	0.0018	0.0020
		TCM1 with C_1		
		TCM with C_2	0.059	0.016
		TCM1 with C_2		
		TCM with C_3	0.018	0.015
		TCM1 with C_3		
		Simulink with C_2	0.020	0.049
		Simulink with C_1		
		Simulink with C_3	0.064	

- (b) Modify the program “do_TCM_8PSK.m” and make a routine “TCM_state_eq2()” so that the TCM system with the encoder of Fig. P9.9.1(b) can be simulated. Run the modified program “do_TCM_8PSK.m” to find the BERs for $E_b/N_0=2$ [dB] and fill in the corresponding blanks of Table P9.9.

```

%do_TCM_8PSK.m
clear
Nf=144; Tf=0.001; Tb=Tf/Nf; % Frame size, Frame and Sample/Bit times
Gc=[1 0 0; 0 5 2]; state_eq='TCM_state_eq1'; Ns=2; % Fig. P9.9.1(a)
%Gc=[1 2 0; 4 1 2]; state_eq='TCM_state_eq2'; Ns=3; % Fig. P9.9.1(b)
[K,N]=size(Gc); Rc=K/N; % Message/Codeword length and Code rate
% Constellation for TCM Simulink block
Constellation2=exp(j*2*pi/8*[0 4 2 6 1 5 3 7]);
% Constellation good for TCM() MATLAB function
Constellation1=exp(j*2*pi/8*[0:7]);
% Constraint length vector
Gc_m=max(Gc,');
for i=1:length(Gc_m), Lc(i)=length(dec2bin(oct2dec(Gc_m(i)))); end
%trel=poly2trellis(Lc,Gc);
Tbdepth=sum(Lc)*5; delay=Tbdepth*K;
Ts=K*Tb; % or N*Rc*Tb % Symbol time
EbN0dBs_t=0:0.1:10; SNRbdBs_t=EbN0dBs_t+3;
BER_theory= prob_error(SNRbdBs_t, 'PSK', K, 'BER');
EbN0dBs=2; % [3 6];
Target_no_of_error=50;
for i=1:length(EbN0dBs)
    EbN0dB=EbN0dBs(i); SNRbdB=EbN0dB+3;
    pemb=TCM(state_eq,K,Ns,N,Constellation1,SNRbdB,Target_no_of_error);
    pembs_TCM8PSK(i)=pemb
    sim('TCM_sim'); pembs_TCM8PSK_sim(i)=BER(1);
end
figure(1), clf
semilogy(EbN0dBs_t,BER_theory,'b'), hold on
semilogy(EbN0dBs,pembs_TCM8PSK_sim,'r*', EbN0dBs,pembs_TCM8PSK,'m*')

```


- (c) For the TCM encoder with no input bit inserted into between the flip-flops (registers), the routines 'TCM()', 'TCM_encoder()', and 'TCM_decoder()' can be modified as 'TCM1()', 'TCM_encoder1()', and 'TCM_decoder1()' (listed below) so that they can accommodate any (convolutional) encoder using the (octal) code generator instead of a function representing the state equation for the TCM encoder. Run the program "do_TCM_8PSK.m" with 'TCM()' replaced by 'TCM1()' and check if the routines work properly. Fill in the corresponding blanks of Table P9.9.
- (d) With reference to the set partitioning shown in Fig. 9.14 (Sec. 9.4.5), why do you think the BER is affected by different constellations? Why is the TCM system of Fig. P9.9.1(b) less affected by different constellations than that of Fig. P9.9.1(a)?

```
function [outputs,state]=TCM_encoder1(Gc,input,state,Constellation)
% generates the output sequence of a binary TCM encoder
% Input:  Gc      = Code generator matrix consisting of octal numbers
%         K       = Number of input bits entering the encoder
%         Nsb     = Number of state bits of the TCM encoder
%         N       = Number of output bits of the TCM encoder
%         input   = Binary input message sequence
%         state   = State of the TCM encoder
%         Constellation = Signal sets for signal mapper
% Output: output = a sequence of signal points on Constellation
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
[K,N]=size(Gc);
for k=1:K, nsb(k)=length(de2bi(oct2dec(max(Gc(k,:)))))-1; end
Nsb=sum(nsb);
tmp= rem(length(input),K);
input= [input zeros(1,(K-tmp)*(tmp>0))];
if nargin<3, state=zeros(1,Nsb);
elseif length(state)==2^N
    Constellation=state; state=zeros(1,Nsb);
end
input_length= length(input);
N_msgsymbol= input_length/K;
input1= reshape(input,K,N_msgsymbol).';
outputs= [];
for l=1:N_msgsymbol
    ub= input1(l,:);
    is=1; nstate=[];
    output=zeros(1,N);
    for k=1:K
        tmp = [ub(k) state(is:is+nsb(k)-1)];
        nstate = [nstate tmp(1:nsb(k))]; is=is+nsb(k);
        for i=1:N
            output(i) = output(i) + ...
                deci2bin1(oct2dec(Gc(k,i)),nsb(k)+1)*tmp';
        end
    end
    state = nstate;
    output = Constellation(bin2deci(rem(output,2))+1);
    outputs = [outputs output];
end
```

```

function decoded_seq=TCM_decoder1(Gc,demod,Constellation,opmode)
% performs the Viterbi algorithm on the PSK demodulated signal
% Inpit: state_eq = External function for state equation saved
%           in an M-file
%           K    = Number of input bits entering encoder at each cycle.
%           Nsb   = Number of state bits of TCM encoder
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
[K,N]=size(Gc);
for k=1:K, nsb(k)=length(de2bi(oct2dec(max(Gc(k,:)))))-1; end
Nsb=sum(nsb);
N_states=2^Nsb;
N_msgsymbol=length(demod);
for m=1:N_states
    for n=1:N_msgsymbol+1
        states(m,n)=0; % inactive in the trellis diagram
        p_state(m,n)=0; n_state(m,n)=0; input(m,n)=0;
    end
end
states(1,1)=1; % make the initial state active
cost(1,1)=0; K2=2^K;
for n=1:N_msgsymbol
    y=demod(n); %received sequence
    n1=n+1;
    for m=1:N_states
        if states(m,n)==1 %active
            xb=dec2bin1(m-1,Nsb);
            for m0=1:K2
                u=dec2bin1(m0-1,K);
                is=1; nstate=[]; output=zeros(1,N);
                for k=1:K
                    tmp = [u(k) xb(is:is+nsb(k)-1)];
                    nstate = [nstate tmp(1:nsb(k))]; is=is+nsb(k);
                    for i=1:N
                        output(i) = output(i) +
                            dec2bin1(oct2dec(Gc(k,i)),nsb(k)+1)*tmp';
                    end
                end
                nxb(m0,:) = nstate;
                yb(m0) = Constellation(bin2deci(rem(output,2))+1);
                nxm0=bin2deci(nxb(m0,:))+1;
                states(nxm0,n1)=1;
                difference=y-yb(m0);
                d(m0)=cost(m,n)+difference*conj(difference);
                if p_state(nxm0,n1)==0
                    cost(nxm0,n1)=d(m0);
                    p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1;
                else
                    [cost(nxm0,n1),i]=min([d(m0) cost(nxm0,n1)]);
                    if i==1, p_state(nxm0,n1)=m; input(nxm0,n1)=m0-1; end
                end
            end
        end
    end
end
end
end
end

```

```

decoded_seq=[];
if nargin<4 | ~strcmp(opmode,'term',4)
    % trace back from the best-metric state (default)
    [min_cost,m]=min(cost(:,n1));
else m=1; %trace back from the all-0 state
end
for n=n1:-1:2
    decoded_seq= [dec2bin1(input(m,n),K) decoded_seq];
    m=p_state(m,n);
end

function [pemb,nombe,notmb]=
    TCM1(Gc,Constellation,SNRbdb,Target_no_of_error)
if nargin<4, Target_no_of_error=50; end
[K,N]=size(Gc); M=2^N; Rc=K/N;
Nf=288; % Number of bits per frame
Nmod=Nf/K; % Number of symbols per modulated frame
SNRb=10.^(SNRbdb/10); SNRbc=SNRb*Rc;
sqrtSNRc=sqrt(2*N*SNRbc); % Complex noise per K=Rc*N-bit symbol
nombe=0; MaxIter=1e6;
for iter=1:MaxIter
    msg=randint(1,Nf); % Message vector
    coded = TCM_encoder1(Gc,msg,Constellation); % TCM encoding
    r= coded + (randn(1,Nmod)+j*randn(1,Nmod))/sqrtSNRc;
    decoded= TCM_decoder1(Gc,r,Constellation);
    nombe = nombe + sum(msg~=decoded(1:Nf)); %
    if nombe>Target_no_of_error, break; end
end
notmb=Nf*iter; % Number of total message bits
pemb=nombe/notmb; % Message bit error probability

```

9.10 DSTBC (Differential Space-Time Block Coding) with Four Transmit Antennas

The DSTBC with two transmit antennas (Sec. 9.4.8) can be extended into the DSTBC with four transmit antennas as follows:

The transmission matrix for sending $[x_0 \ x_1 \ x_2 \ x_{3+0} \ x_{3+1} \ x_{3+2} \ \cdots x_{3n} \ x_{3n+1} \ x_{3n+2} \ \cdots]$ to the receiver is constructed as

$$S_{4(n+1)} = S_{4n} \cdot X_{3n} \quad \text{with} \quad S_0 = I_{4 \times 4} \quad (\text{P9.10.1})$$

where

$$X_{3n} = \text{Space} \downarrow \begin{array}{c} \xrightarrow{\text{Time}} \\ \begin{bmatrix} x_{3n} & -x_{3n+1}^* & x_{3n+2}^*/\sqrt{2} & x_{3n+2}^*/\sqrt{2} \\ x_{3n+1} & x_{3n}^* & x_{3n+2}^*/\sqrt{2} & -x_{3n+2}^*/\sqrt{2} \\ x_{3n+2}/\sqrt{2} & x_{3n+2}/\sqrt{2} & -x_{3n,R} + jx_{3n+1,I} & -x_{3n+1,R} + jx_{3n,I} \\ x_{3n+2}/\sqrt{2} & -x_{3n+2}/\sqrt{2} & x_{3n+1,R} + jx_{3n,I} & -x_{3n,R} - jx_{3n+1,I} \end{bmatrix} \frac{1}{\sqrt{3}} \end{array} \quad (\text{P9.10.2})$$

Each row of this transmission matrix is transmitted through each antenna and each column is sent during each one of four consecutive periods. Then the signal received through a noise-free channel having frequency response $\mathbf{h}=[h_1 \ h_2 \ h_3 \ h_4]$ can be described as

$$R_{4(n+1)} = \begin{bmatrix} r_{4(n+1)} & r_{4(n+1)+1} & r_{4(n+1)+2} & r_{4(n+1)+3} \end{bmatrix} = \mathbf{h} S_{4(n+1)} \stackrel{(\text{P9.10.1})}{=} \mathbf{h} S_{4n} X_{4n} = R_{4n} X_{4n} \quad (\text{P9.10.3})$$

The receiver uses the following decoding rule^[W-10] to estimate the message sequence $\{x_n\}$:

$$\hat{\mathbf{x}}_{3n} = \begin{bmatrix} \hat{x}_{3n} \\ \hat{x}_{3n+1} \\ \hat{x}_{3n+2} \end{bmatrix} = \frac{\sqrt{3}}{|r_{4n}|^2 + |r_{4n+1}|^2 + |r_{4n+2}|^2 + |r_{4n+3}|^2} \times \begin{bmatrix} r_{4n}^* r_{4(n+1)} + r_{4n+1}^* r_{4(n+1)+1} + ((r_{4n+2}^* - r_{4n+3}^*)(-r_{4(n+1)+2} + r_{4(n+1)+3}) - (r_{4n+2} + r_{4n+3})(r_{4(n+1)+2}^* + r_{4(n+1)+3}^*)) / 2 \\ r_{4n+1}^* r_{4(n+1)} - r_{4n}^* r_{4(n+1)+1} + ((r_{4n+2}^* - r_{4n+3}^*)(r_{4(n+1)+2} + r_{4(n+1)+3}) + (r_{4n+2} + r_{4n+3})(-r_{4(n+1)+2}^* + r_{4(n+1)+3}^*)) / 2 \\ (r_{4n+2}^*(r_{4(n+1)} + r_{4(n+1)+1}) + r_{4n+3}^*(r_{4(n+1)} - r_{4(n+1)+1}) + (r_{4n} + r_{4n+1})r_{4(n+1)+2}^* + (r_{4n} - r_{4n+1})r_{4(n+1)+3}^*) / \sqrt{2} \end{bmatrix} \quad (\text{P9.10.4})$$

Complete the following program “test_DSTBC_H4_PSK.m” and run it to see if this scheme works.

```
%test_DSTBC_H4_PSK.m
clear, clf
ITR=10; MaxIter=100;
sq2= sqrt(2); sq3= sqrt(3); sq6=sqrt(6);
M=4; phs= 2*pi*[0:M-1]/M; MPSKs= exp(j*phs); % 4-PSK symbols
h=[0.95*exp(j*0.5) exp(-j*0.4) 1.02*exp(-j*0.2) 0.98*exp(j*0.1)]; % Channel
dh= 0.01*[exp(-j*0.1) -exp(j*0.2) exp(j*0.1) -exp(-j*0.2)]; % Channel variation
Amp_noise = 0.2; % Amplitude of additive Gaussian noise
ner=0;
for itr1=1:ITR
    S= eye(4); % S0=I;
    r= h*S + Amp_noise*[randn(1,4)+j*randn(1,4)]; % Eq.(P9.10.3)
    x=[]; xh=[]; % Initialize the sequence of transmitted signal
    nn=[-3:0];
    for n=1:MaxIter
        nn= nn+4;
        xn= MPSKs(randint(1,3,M)+1); % +-1+-j
        x= [x xn]; % Sequence of transmitted symbols
        x1=xn(1)/sq3; x2=xn(2)/sq3; x3=xn(3)/sq6; x3c=x3';
        x1R=real(x1); x1I=imag(x1); x2R=real(x2); x2I=imag(x2);
        X=[x1 -x2' x3c x3c; x2 x1' x3c -x3c;
            x3 x3 -x1R+j*x2I x2R+j*x1I; x3 -x3 -x2R+j*x1I -x1R-j*x2I]; %Eq.(P9.10.2)
        S = S*X; % Encoded signal Eq.(P9.10.1)
        r0=r; % Previously received signal
        rs = (h+dh*n)*S; % Received signal Eq.(P9.10.3)
        noise = Amp_noise*[randn(1,4)+j*randn(1,4)]; % Noise components
        r = rs + noise; % Received signal
        r034_=(r0(?) -r0(?))'; r034_r3=r034_*r(3); r034_r4=r034_*r(4);
        r034_ = r0(?) +r0(?); r034r3=r034_*r(3)'; r034r4=r034_*r(4)';
        xhn1= r0(?)'*r(?) +r0(2)*r(2)' + (-r034_r3+r034_r4-r034r3-r034r4)/2;
        xhn2= r0(2)'*r(?) -r0(?)*r(2)' + (r034_r3+r034_r4-r034r3+r034r4)/2;
        xhn3= (r0(3)'*(r(?) +r(2)) +r0(4)'*(r(?) -r(2)) ...
            +(r0(?) +r0(2))*r(3)' + (r0(?) -r0(2))*r(4)')/sq2;
        xhns= [xhn1 xhn2 xhn3]*(sq3/(r0*r0'))'; % Decoded signal Eq.(P9.10.4)
        xhn= PSK_slicer(xhns,M);
        xh= [xh xhn];
    end
    ner= ner + sum(abs(x-xh)>0.1);
end
SER = ner/(MaxIter*3*ITR) % Symbol error rate
```

- Accordingly, we can use the complex convolution or modulation property (1.4.11) of CTFT to find the spectrum of a sinusoidal pulse $r_D(t) \cos(\omega_k t)$ of frequency $\omega_k = 2\pi k/T_B$ and duration $D=T_B$ as

$$\begin{aligned}
 r_D(t) \cos(\omega_k t) &\xrightarrow[(1.4.11)]{(11.2.1), (11.2.2)} \frac{1}{2\pi} \pi \{ \delta(\omega - \omega_k) + \delta(\omega + \omega_k) \} * T_B \frac{\sin(\omega T_B/2)}{\omega T_B/2} \\
 &= \frac{T_B}{2} \left\{ \frac{\sin((\omega - \omega_k) T_B/2)}{(\omega - \omega_k) T_B/2} + \frac{\sin((\omega + \omega_k) T_B/2)}{(\omega + \omega_k) T_B/2} \right\} = \frac{T_B}{2} \{ \text{sinc}((f - f_k) T_B) + \text{sinc}((f + f_k) T_B) \}, f_k = \frac{k}{T_B}
 \end{aligned} \tag{11.2.3}$$

Fig. 11.3(b) based on these facts shows that the bandwidth of an N -subcarrier DFT-based OFDM signal is $(N+1)/T_B$ [Hz] where N is the DFT size. If each carrier is loaded with b bits of data, the data transmission bit rate is

$$\frac{Nb}{T_B + T_g} \text{ [bits/s]} \quad (T_B : \text{block interval, } T_g : \text{guard interval, i.e., cyclic prefix duration}) \tag{11.2.4}$$

so that the bandwidth efficiency is

$$\frac{Nb/(T_B + T_g)}{(N+1)/T_B} = \frac{Nb}{N+1} \frac{T_B}{T_B + T_g} \text{ [bits/s/Hz]} \tag{11.2.5}$$

which becomes closer to the bandwidth efficiency b of the single-carrier communication as the number N of OFDM subcarriers increases and the guard interval T_g gets shorter.

11.3 CARRIER RECOVERY AND SYMBOL SYNCHRONIZATION

This topic was discussed in Chapter 8. To address the issue of carrier recovery and symbol timing on OFDM systems, let us begin with observing the effects of *STO* (*symbol time offset*) and *CFO* (*carrier frequency offset*). With this objective, we modify the MATLAB program “do_OFDM0.m” (in Section 11.1) by increasing the value of SNRbdB (in the 19th line) to 20 and activating the 11th line so that the STO amounting to one sample time is introduced. Running the modified program will yield a received signal constellation diagram like Fig. 11.4(a) and a much worse BER than that with neither STO nor CFO. Also, to take a look at the effect of CFO, activate the 14th line from the bottom (with the 11th line inactivated and SNRbdB=20) so that the CFO of 0.1 is introduced. This will yield a received signal constellation diagram like Fig. 11.4(b) and a worse BER. These effects of STO and CFO, called the *ISI* (*inter-symbol interference*) and *ICI* (*inter-carrier interference*), on the received signal can be observed and analyzed through the following expression of a time-domain received OFDM symbol:

$$y^{(l)}[n] = \frac{1}{N} \sum_{k=0}^{N-1} G_k X_k^{(l)} e^{j2\pi(k+\varepsilon)(n+\delta)/N} + \text{noise} \tag{11.3.1}$$

where $y^{(l)}[n]$ denotes a (time-domain) received sequence for the l^{th} OFDM symbol, G_k the channel frequency response (at the k^{th} frequency index), $X_k^{(l)}$ the l^{th} (frequency-domain) transmitted OFDM symbol, ε the CFO, δ the STO, and N the DFT size.

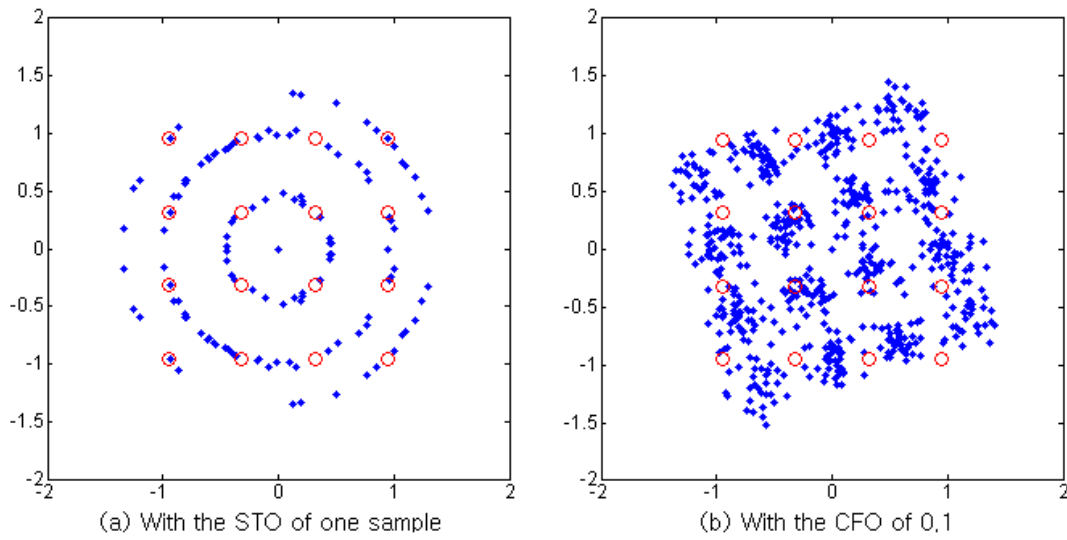
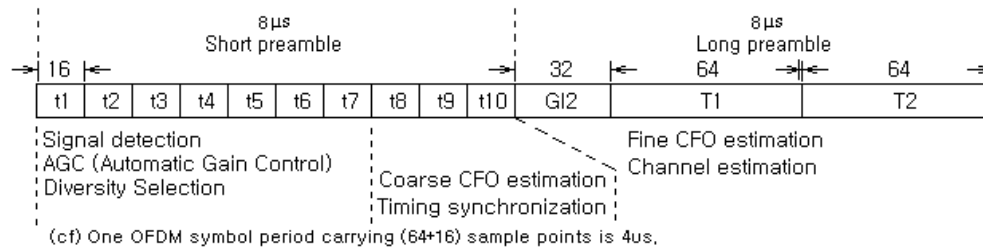


Figure 11.4 The signal constellation diagrams showing the effects of STO and CFO



(a) The structure of preamble in the IEEE Standard 802.11a

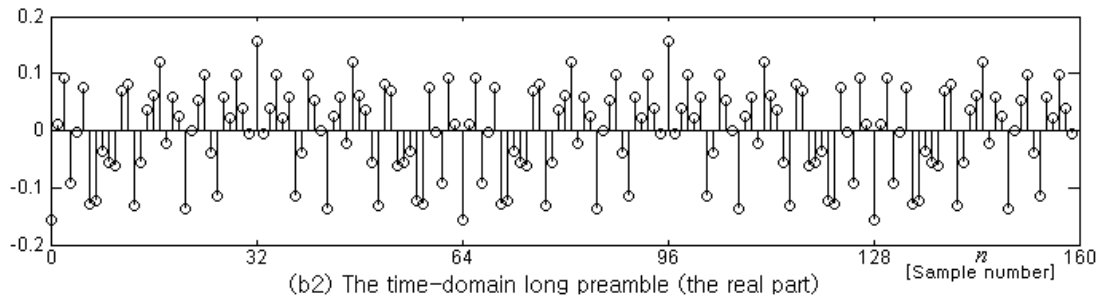
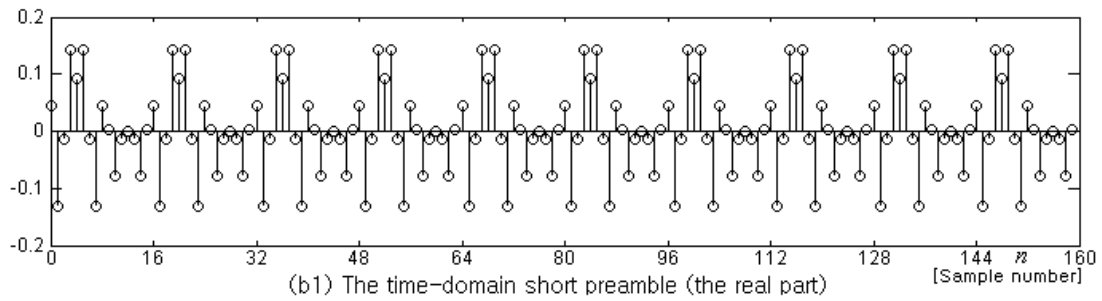


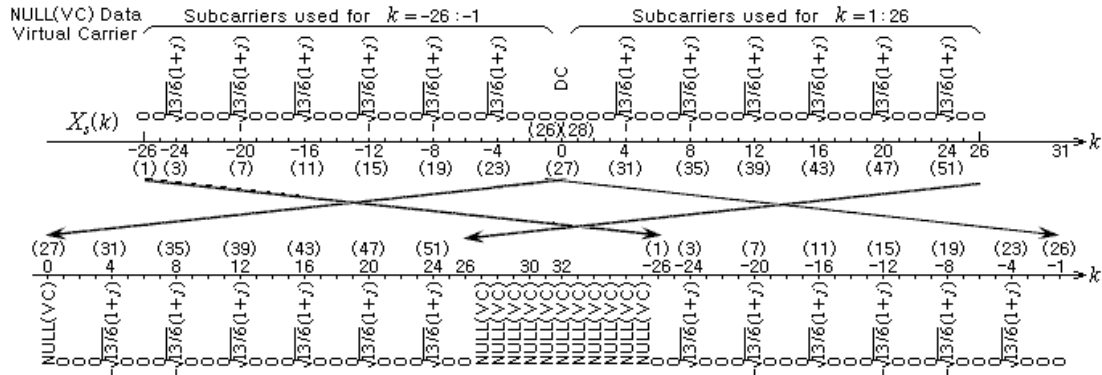
Figure 11.5 The time-domain preambles in the IEEE Standard 802.11a

```

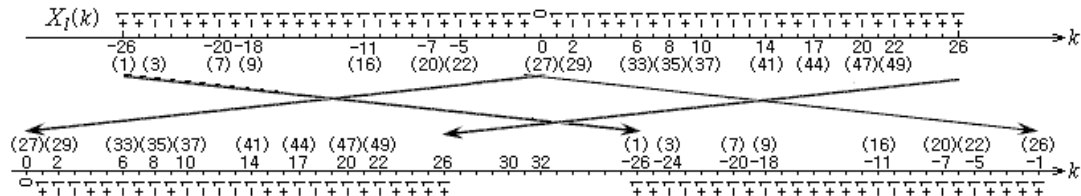
function [short_preamble,Xs]=short_train_seq(Nfft)
if nargin<1, Nfft=64; end
sqrt136 = sqrt(13/6)*(1+i); Xs=zeros(1,53);
Xs([3 11 23 39 43 47 51]) = sqrt136; % Frequency domain for k=-26:26
Xs([7 15 19 31 35]) = -sqrt136;
xs = ifft([Xs(27:53) zeros(1,11) Xs(1:26)]); % Time domain
short_preamble = [xs xs xs(1:Nfft/2)];

function [long_preamble,Xl]=long_train_seq(Nfft)
if nargin<1, Nfft=64; end
Xl([1:26 28:53]) = 1; % Frequency domain for k=-26:26
Xl([3 4 7 9 16 17 20 22 29 30 33 35 37:41 44 45 47 49])=-1;
xl = ifft([Xl(27:53) zeros(1,11) Xl(1:26)]); % Time domain
long_preamble= [xl(Nfft/2+1:Nfft) xl xl];

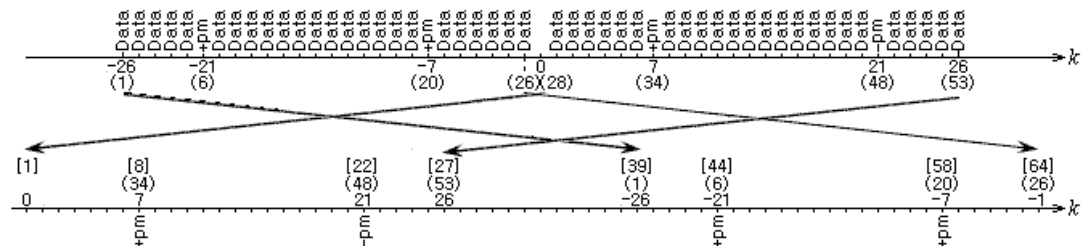
```



(a) Frequency-domain short preamble displayed in the negative-positive/positive-negative frequency axes



(b) Frequency-domain long preamble displayed in the negative-positive/positive-negative frequency axes



(c) Pilot symbol displayed in the negative-positive/positive-negative frequency axes

Figure 11.6 Short preamble, long preamble, and pilot symbol in the IEEE 802.11a standard

```

function phase_estimate=phase_from_pilot(outofFFT,pm)
% To estimate the phase offset based on the received pilot symbol
% outofFFT: OFDM symbol obtained from FFT at the RCVR
% pm      : the sign of pilot signals for each OFDM symbol
%          determined by the PN sequence
phase_estimate = angle(outofFFT([8 22 44 58]) * [pm;-pm;pm;pm]) ;

function y=compensate_phase(x,phase)
% To compensate the received OFDM symbol x by phase deviation
y = x*exp(-j*phase) ;

```

Many carrier recovery schemes to overcome the carrier phase distortion and frequency offset and many symbol synchronization schemes to overcome the STO effect are implemented with the correlator, PLL, narrow-band filter, and/or tuner using the pilot symbol, phase reference symbol, training symbol (called a *preamble*), cyclic prefix (CP), and/or null symbol. For example, Fig. 11.5(a) shows the structure of the short and long OFDM training symbols (preambles) that is used in the IEEE Standard 802.11a[W-8, Sec. 17.3.3] where the short preamble consists of 10 repetitions of a (16-sample) ‘short training sequence’ and the long one consists of 2.5 repetitions of a (64-sample) ‘long training sequence’. The above MATLAB routines ‘short_train_seq()’ and ‘long_train_seq()’ can be used to generate the short and long preambles in the time/frequency domain, respectively. Figs. 11.5(b1) and (b2) show the real parts of the short and long preambles that have been generated using each of the two routines, respectively. Figs. 11.6(a) and (b) show the frequency-domain short and long preambles, respectively, from which it can be seen that the energy of the frequency-domain long preamble $X_l(k)$ is 52 and the frequency-domain short preamble $X_s(k)$ has nonzero values of $\sqrt{13/6}(1+j)$ only for 12 subcarriers so that its energy is also $12 \times (\sqrt{13/6}|1+j|)^2 = 52$. On the other hand, Fig. 11.6(c) shows that the pilot symbols (in the IEEE Standard 802.11a) are inserted in the $\{8^{\text{th}}, 22^{\text{th}}, 44^{\text{th}}, 58^{\text{th}}\}$ subcarriers (each corresponding to frequency indices $k = \{7, 21, -21, -7\}$, respectively,) of OFDM symbols. The above routines ‘phase_from_pilot()’ and ‘compensate_phase()’ estimate the phase offset based on the received pilot symbols and compensate the phase of the received signal by the phase estimate, respectively.

Not only the phase compensation but also the CFO (carrier frequency offset) compensation is needed for successful carrier recovery. In the IEEE Standard 802.11a, the fixed-lag correlation of the short training sequence is used for the coarse estimation of CFO and the fixed-lag correlation of the long training sequence is used for the fine estimation of CFO as follows:

$$\text{Coarse CFO estimate: } \hat{\varepsilon}_c = \frac{N}{2\pi N_g} \angle \left\{ \sum_{n=1}^{N_g} x_s[n+N_g] x_s^*[n] \right\} \quad (11.3.2a)$$

$$\text{Fine CFO estimate: } \hat{\varepsilon}_f = \frac{1}{2\pi} \angle \left\{ \sum_{n=1}^N x_l[n+N] x_l^*[n] \right\} \quad (11.3.2b)$$

where x_s is the (time-domain) short preamble, x_l the (time-domain) long preamble, N the FFT size and also the period of x_l , and N_g the length of the guard interval or cyclic prefix and also the period of x_s . Note that the estimable CFO ranges of the coarse and fine CFO estimations are

$$\frac{N}{2\pi N_g} [-\pi, +\pi] = \frac{64}{2\pi \times 16} [-\pi, +\pi] = [-2, +2] \quad \text{and} \quad \frac{1}{2\pi} [-\pi, +\pi] = [-0.5, +0.5] \quad (11.3.3)$$

respectively, where the latter estimate can be represented with less error by the same number of bits.

A question about the phase estimator (11.3.2) may arise in your mind. How can the information about the frequency offset be extracted from the time-domain correlation of the signal? To find the answer to this question, let us use Eq. (11.1.3) to write the values of, say, the short preamble $x_s[n]$ at two points apart from each other by $N_g=16$ samples as

$$x_s[n] \stackrel{(11.1.3)}{=} \frac{1}{N} \sum_{m=0}^{16-1} X_s(4m) e^{j2\pi(4m+\varepsilon)n/64} \quad \text{with CFO of } \varepsilon$$

$$\text{and } x_s[n+16] \stackrel{(11.1.3)}{=} \frac{1}{N} \sum_{m=0}^{16-1} X_s(4m) e^{j2\pi(4m+\varepsilon)(n+16)/64} \quad (11.3.4)$$

where we have taken into consideration that the spectrum $X_s(k)$ of the short preamble $x_s[n]$ has nonzero values only for $k=4m$ (see Fig. 11.6(a)). Multiplying one with the other's conjugate yields

$$x_s[n+16]x_s^*[n] = \frac{1}{N} \sum_{m=0}^{16-1} X_s(4m) e^{j2\pi(4m+\varepsilon)(n+16)/64} \frac{1}{N} \sum_{i=0}^{16-1} X_s^*(4i) e^{-j2\pi(4i+\varepsilon)n/64}$$

(The cross multiplication terms for $m \neq i$ disappears due to their orthogonality.)

$$= \frac{1}{N^2} \sum_{m=0}^{16-1} X_s(4m) X_s^*(4m) e^{j2\pi(4m+\varepsilon)(n+16)/64 - j2\pi(4m+\varepsilon)n/64}$$

$$= \frac{16 \times 13}{6N^2} (1+j)(1-j) e^{j2\pi(4m+\varepsilon)16/64} = \frac{13}{12 \times 64} \angle(2m\pi + \frac{\pi}{2}\varepsilon) = K \angle \frac{\pi}{2} \varepsilon \quad (11.3.5)$$

which presents the basis of the coarse CFO estimator Eq. (11.3.2a). Similarly, the correlation of the long preamble yields

$$x_l[n+N]x_l^*[n] = \frac{1}{N} \sum_{m=0}^{N-1} X_l(m) e^{j2\pi(m+\varepsilon)(n+N)/N} \frac{1}{N} \sum_{i=0}^{N-1} X_l^*(i) e^{-j2\pi(i+\varepsilon)n/N}$$

(The cross multiplication terms for $m \neq i$ disappears due to their orthogonality.)

$$= \frac{1}{N^2} \sum_{m=0}^{N-1} X_l(m) X_l^*(m) e^{j2\pi(m+\varepsilon)(n+N)/N - j2\pi(m+\varepsilon)n/N}$$

$$= \frac{1}{N} e^{j2\pi(m+\varepsilon)} = \frac{1}{N} \angle(2m\pi + 2\pi\varepsilon) = \frac{1}{N} \angle 2\pi\varepsilon \quad (11.3.6)$$

which presents the basis of the fine CFO estimator Eq. (11.3.2b). Note from Eqs. (11.3.2a) and (11.3.2b) that the phase is taken for the sum of the multiplication terms instead of the average. Why? Because the phase of a complex number does not depend on its magnitude. That is also why the average is not taken for the sum of the four pilot symbol terms in the routine 'phase_from_pilot()'.

The following routines 'coarse_CFO_estimate()' and 'fine_CFO_estimate()' can be used to get the coarse and fine CFO estimates via Eqs. (11.3.2a) and (11.3.2b), respectively. Note that the two routines differ in the frequency range to estimate, but not in the accuracy and therefore we can get a good CFO estimate using only the first one. However, to emphasize the quantitative difference between the two routines, it is assumed that an 8-bit number representation with relative resolution 2^{-8} is used for storing the CFO estimation result. The next MATLAB program "do_CFO.m" uses the routines 'short_train_seq()'/'long_train_seq()' to generate the short/long preambles, uses the routine 'set_CFO()' to set up a CFO, uses 'coarse_CFO_estimate()' and 'fine_CFO_estimate()' to estimate the CFO, and then uses 'compensate_CFO()' to compensate the CFO.

```

function coarse_CFO_est = coarse_CFO_estimate(tx,NB,Nw,STO)
% Input:
%   tx = Received signal
%   NB = Which block of size Nw to start computing correlation with?
%   Nw = Correlation window size
%   STO= Symbol Time Offset
% Output: coarse_CFO_est = Estimated carrier frequency offset
if nargin<4, STO = 0; end
if nargin<3, Nw = 16; end
if nargin<2, NB = 6; end
for i=1:2
    nn = STO + Nw*(NB+i) + [1:Nw];
    CFO_est(i) = angle(tx(nn+Nw)*tx(nn)'); % Eq.(11.3.2a)
end
CFO_est = sum(CFO_est)/pi; % Average
coarse_CFO_est = CFO_est - mod(CFO_est,4/128); % Stored with 8 bits

```

```

function fine_CFO_est = fine_CFO_estimate(tx,coarse_CFO_est,Nfft,STO)
% Input : tx = Received signal
%         coarse_CFO_est = coarse CFO estimate
%         Nfft = FFT size
%         STO = Symbol Time Offset
% Output: fine_CFO_est = Estimated carrier frequency offset
if nargin<4, STO = 0; end
if nargin<3, Nfft = 64; end
if nargin<2, coarse_CFO_est = 0; end
tx1 = CFO_compensation(tx,coarse_CFO_est,Nfft,STO);
nn = STO + Nfft/2 + [1:Nfft];
cfo_est = angle(tx1(nn+Nfft)*tx1(nn)')/(2*pi); % Eq.(11.3.2b)
fine_CFO_est = CFO_est - mod(CFO_est,1/128); % Stored with 8 bits

```

```

%do_CFO.m
clear, clf
Nfft = 64; Ng = 16;
CFO = 1.7; phase = 0; STO = -1; % CFO/Phase Offset/STO
NB = 6; % Which block to start computing the correlation with?
Nw = Ng; % Correlation window size
[short_preamble,S] = short_train_seq(Nfft);
[long_preamble,L] = long_train_seq(Nfft);
% Time-domain training symbol
tx = [short_preamble long_preamble];
% Set up a pseudo CFO
tx_offset = set_CFO(tx,CFO,phase,Nfft);
% Coarse CFO estimation
coarse_CFO = coarse_CFO_estimate(tx_offset(1:160),NB,Nw,STO);
% Fine CFO estimation
fine_CFO = fine_CFO_estimate(tx_offset(161:320),coarse_CFO,Nfft);
% Overall CFO estimate
CFO_estimate = coarse_CFO + fine_CFO;
form1 = '\n For CFO=%10.8f, CFO estimate(%10.8f) = ';
form2 = 'coarse estimate(%10.8f)+fine estimate(%10.8f)\n';
fprintf([form1 form2],CFO,CFO_estimate,coarse_CFO,fine_CFO);
% CFO compensated symbols
tx_compensated = compensate_CFO(tx_offset,CFO_estimate,Nfft,STO);
discrepancy = norm(tx-tx_compensated)/length(tx)

```

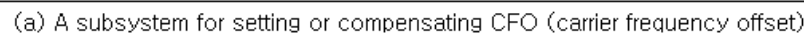


Figure 11.7 Subsystems to be used for CFO estimation and compensation

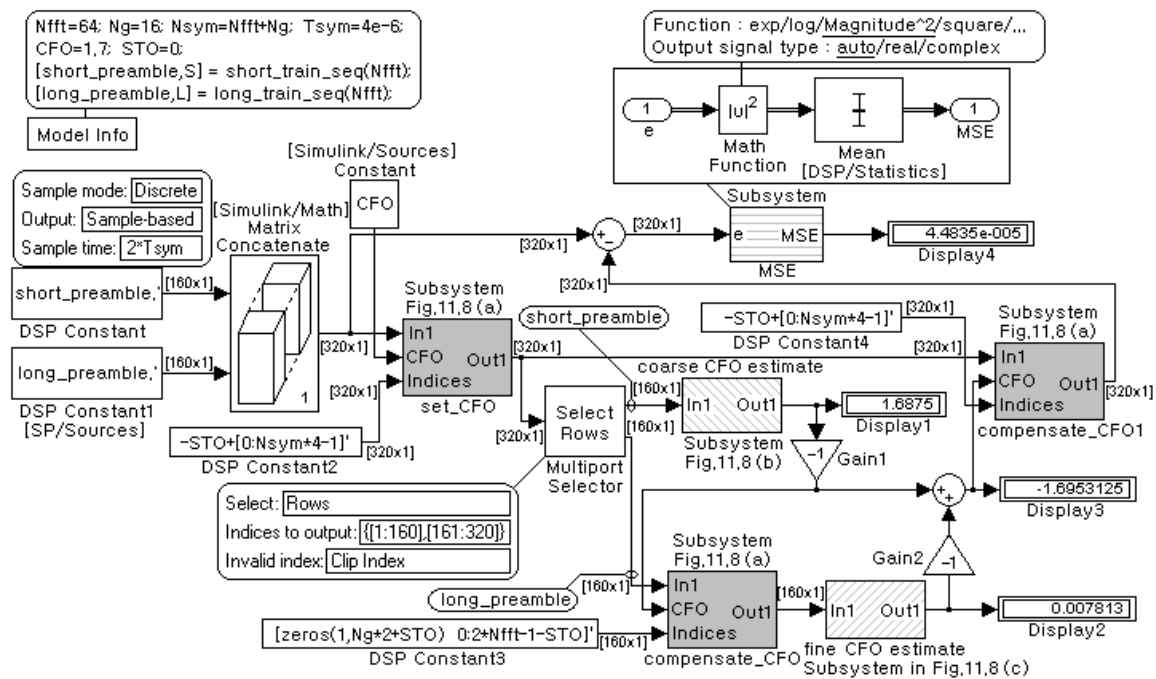


Figure 11.8 Simulink model for demonstrating CFO estimation and compensation ("CFO_sim.mdl")

Fig. 11.7 shows the three Simulink subsystems that can be used to estimate and compensate the CFO in a Simulink model for OFDM system simulation. Note the following about the subsystems:

- The subsystem of Fig. 11.7(a) is equivalent to the MATLAB routine 'set_CFO()', but it can be made to work like 'compensate_CFO()' by negating the CFO input as can be seen in Fig. 11.8.
- The subsystem of Fig. 11.7(b) is equivalent to the MATLAB routine 'coarse_CFO_estimate()', which uses Eq. (11.3.2a) to make a coarse CFO estimation based on the two correlation values, one between the 8th (16-sample) block and the 9th block and the other between the 9th block and the 10th block of the received sequence corresponding to the (160-sample) short preamble.
- The subsystem of Fig. 11.7(c) is equivalent to 'fine_CFO_estimate()', which uses Eq. (11.3.2b) to make a fine CFO estimation based on the correlation value between the 1st (64-sample) block and the 2nd block of the received sequence corresponding to the (160-sample) long preamble.

Fig. 11.8 shows a Simulink model named "CFO_sim.mdl", which demonstrates the CFO estimation and compensation using the short and long preambles and the three subsystems depicted in Fig. 11.7. About this Simulink model, note the following:

- The coarse CFO estimate is computed by the (subsystem) block 'coarse CFO estimate' and then is used to compensate the CFO of the long preamble through the block 'compensate_CFO'.
- The fine CFO estimate (obtained from the correlation value of the coarse-CFO-compensated long preamble by the block 'fine CFO estimate') and the coarse CFO estimate are negated, fed into the block 'compensate_CFO1', and then used to compensate the CFO of a general signal.

Interested readers are invited to compose the Simulink model "CFO_sim.mdl" and run it to see if the CFO is well tracked for different values of CFO and consequently, the mean square error between the original signal and the CFO-inserted-and-compensated one (computed by the block 'MSE') is very small.

```

%do_STO_estimation.m
% To estimate the STO (Symbol Time Offset)
%Copyright: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
clear, clf
Cor_thd = 0.93; % Threshold of correlation for peak detection
Nfft=64; Ng=16; % FFT/Guard interval size
Nsym=Nfft+Ng; Nw=Ng; % Symbol/Window size
Nd=65; % Remaining period of the last symbol in the previous frame
% Make a pseudo frame to transmit
t_frame = rand(1,Nd)-0.5+j*(rand(1,Nd)-0.5);
N_Symbols = 3; % Number of OFDM symbols to be generated for simulation
for i=1:N_Symbols
    symbol = rand(1,Nfft)-0.5+j*(rand(1,Nfft)-0.5); % An arbitrary data
    symbol_cp = [symbol(end-Ng+1:end) symbol]; % OFDM Symbol with CP
    t_frame = [t_frame symbol_cp]; % Append a frame by a symbol with CP
end
L_frame = length(t_frame); nn=0:L_frame-1;
noise = 0.2*(rand(1,L_frame)-0.5 +j*(rand(1,L_frame)-0.5));
r = t_frame + noise;
sig_w = zeros(2,Nw); % Initialize the two sliding window buffers
STOs = [0]; % Initialize the STO buffer
for n=1:L_frame
    sig_w(1,:) = [sig_w(1,2:end) r(n)]; % Update signal window 1
    m = n-Nfft;
    if m>0
        sig_w(2,:)=[sig_w(2,2:end) r(m)]; % Update signal window 2
        den = norm(sig_w(1,:))*norm(sig_w(2,:));
        corr(n) = abs(sig_w(1,:)*sig_w(2,:)')/den;
        if corr(n)>Cor_thd & m>STOs(end)+Nsym-15
            STOs=[STOs m]; % List the estimated STO
        end
    end
end
end
Estimated_STOs = STOs(2:end)
True_STOs = Nd+Ng + [0:N_Symbols-1]*Nsym
subplot(311)
stem(nn,real(r)), ylim([-0.6 1.1])
hold on, stem(True_STOs,0.8*ones(size(True_STOs)),'k*')
stem(Estimated_STOs,0.6*ones(size(Estimated_STOs)),'rx')
title('Estimated Starting Times of OFDM Symbols')
subplot(312)
plot(nn+1,corr), ylim([0 1.2]), hold on,
stem(Estimated_STOs+Nfft,corr(Estimated_STOs+Nfft),'r:^')
% The points at which the correlation is presumably maximized,
% yielding the STO estimates.
stem(Estimated_STOs,0.9*ones(size(Estimated_STOs)),'rx')
title('Correlation between two sliding windows across Nfft samples')
set(gca,'XTick',sort([Estimated_STOs Estimated_STOs+Nfft]))

```

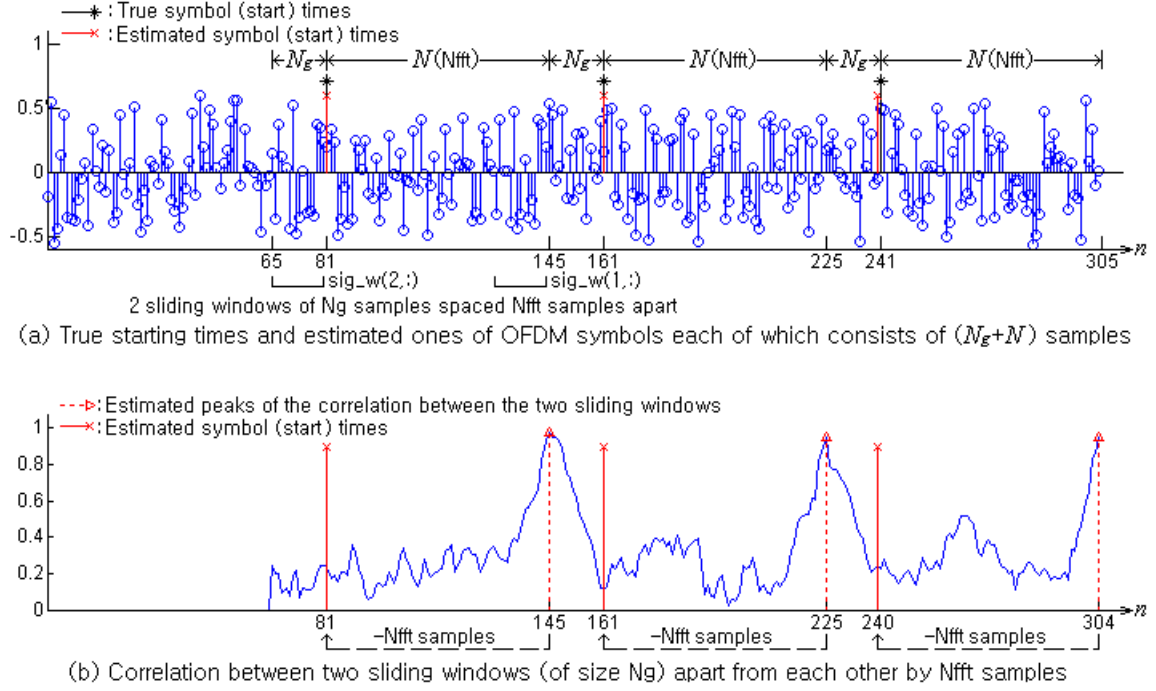


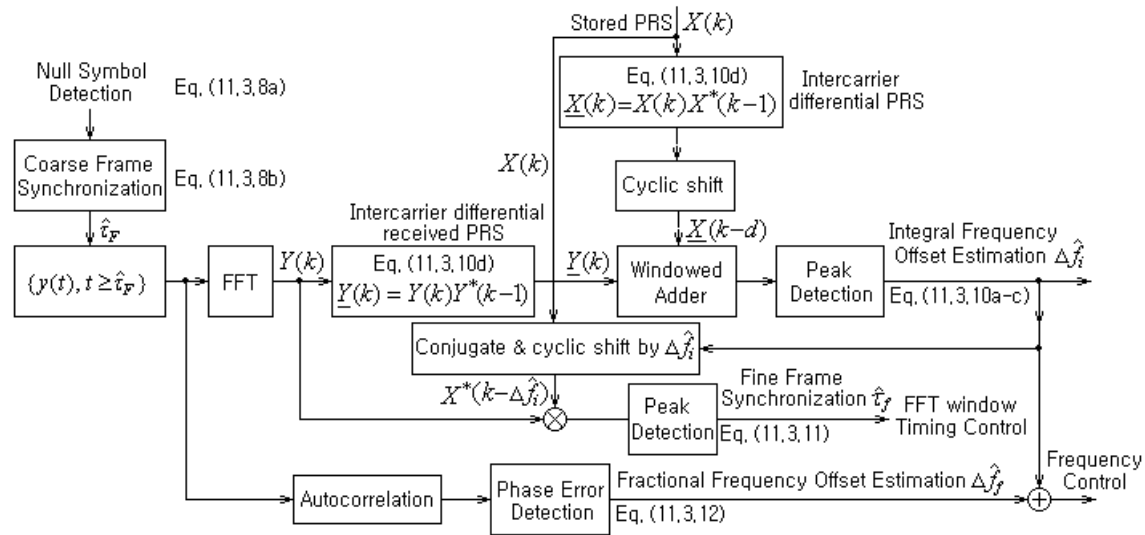
Figure 11.9 STO (symbol time offset) estimation

The above program “do_STO_estimation.m” demonstrates how the correlation value can be used to estimate the STO of an OFDM symbol consisting of N_g (guard interval size) cyclic prefix (CP) samples and N (FFT size) signal samples. As shown in Fig. 11.9(a), it keeps updating a $2 \times N_g$ matrix ‘sig_w’ that contains the N_g samples of two sliding windows in each of its two rows, finds the (local) peak times of the crosscorrelation value between two row vectors $\mathbf{x}_{1,n} = x(n - N_g + 1 : n)$ and $\mathbf{x}_{2,n} = x(n - N - N_g + 1 : n - N)$ (corresponding to two sliding windows), and sets the times ($m = n - N_{fft}$) of N samples before the local peak times to the STO estimates (to be stored in the vectors named ‘STOs’ and ‘Estimated_STOs’). In Fig. 11.9(b), the measured peak times of the correlation value and the corresponding STO estimates are denoted by dotted lines and solid lines, respectively. Note that the on-line detection of local maxima or minima of a noisy signal is not so simple since there may be spurious peaks around true peaks due to the noise. That is why the correlation value is divided by the product of the norms of the two vectors for normalization as

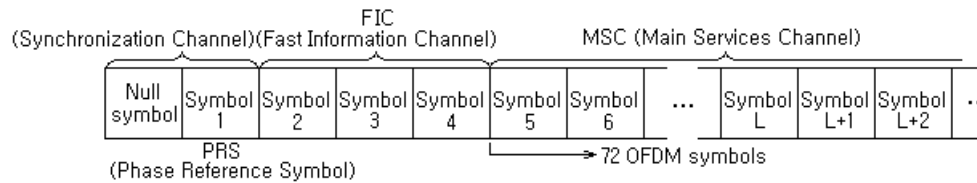
$$\phi_y[n] = \frac{\left| \sum_{m=n-N_g+1}^n x[m]x^*[m-N] \right|}{\sqrt{\sum_{m=n-N_g+1}^n |x[m]|^2} \sqrt{\sum_{m=n-N_g+1}^n |x[m-N]|^2}} = \frac{|\text{sig_w}(1,:) \text{sig_w}(2,:)|}{\|\text{sig_w}(1,:)\| \|\text{sig_w}(2,:)\|} \quad (11.3.7)$$

(‘(apostrophe): the MATLAB operator representing the conjugate transpose, i.e., Hermitian)

and a threshold of, say, $\text{Cor_thd} = 0.93$ is used to determine if the presumable peak time has been reached or not. The additional condition ($m > \text{STOs}(\text{end}) + N_{sym} - 15$) to determine peak times has been required not to let another (most probably wrong) peak reported within $N_{sym} - 15$ ($N_{sym} = N_g + N$: OFDM symbol duration) samples after a peak is reported.



(a) Block diagram of the synchronization scheme for the EUREKA-147 DAB system



(b) The EUREKA-147 transmission frame structure of 96ms for transmission mode I

Figure 11.10 The synchronization scheme and transmission frame of the EUREKA-147 DAB system

Table 11.1 Transmission modes of the EUREKA-147 DAB system

Transmission mode	I (Large area)	II (Topographical situation)	III (Satellite transmission)	IV (Limited speed and direct line of sight)
Maximum carrier frequency	375 MHz	1.5 GHz	3 GHz	1.5 GHz
Number of subcarriers	1,536	384	192	768
Subcarrier interval frequency	1 kHz	4 kHz	8 kHz	2 kHz
Guard interval time	246 us	62 us	31 us	123 us
OFDM symbol period	1 ms	250 us	125 us	500 us
Frame period	96 ms	24 ms	24 ms	48 ms

Now, let us consider the synchronization scheme that is used in the EUREKA-147 standard for DAB (digital audio broadcasting). It consists of the four processes working in coordination, i.e. the coarse frame synchronization, the IFO (integral frequency offset) estimation, the fine frame synchronization, and the FFO (fractional frequency offset) estimation, as depicted by the block diagram of Fig. 11.10(a). Fig. 11.10(b) shows the EUREKA-147 transmission frame structure of 96ms for transmission mode I where the null symbol and *PRS* (*phase reference symbol*) symbol preceding the frame can be used for the frame synchronization.

1. Coarse Frame Synchronization

The tactics to estimate the starting points of a null symbol and a frame based on the energy ratio between the two adjacent windows of size N_w can be expressed by the following equations:

$$\text{Null time estimate: } \hat{n}_N = \text{Arg Min}_{n-N_w} \frac{\sum_{m=n-N_w+1}^n r_m^* r_m}{\sum_{m=n-N_w+1}^n r_{m-N_w}^* r_{m-N_w}} \quad (11.3.8a)$$

$$\text{Frame time estimate: } \hat{n}_F = \text{Arg Max}_{n-N_w} \frac{\sum_{m=n-N_w+1}^n r_m^* r_m}{\sum_{m=n-N_w+1}^n r_{m-N_w}^* r_{m-N_w}} \quad (11.3.8b)$$

The estimated null symbol period ($\hat{n}_F - \hat{n}_N$) can be used to detect the transmission mode since it depends on the transmission mode (see Table 11.1).

The following program “do_sync_w_double_window.m” simulates the process of catching the starting points of a null symbol and the frame (preceded by the null symbol) based on the energy ratio between two successive blocks of size N_{null} . It also simulates the process of estimating the starting points of each OFDM symbol based on the correlation value between two blocks of size N_g spaced N_{fft} samples apart (N_{fft} : FFT size). To minimize the number of computations as well as to save the memory, we maintain several windows (buffers) to store some duration of signal samples, powers, energies, and correlations as follows: (see Fig. 11.11)

- (1) $\text{power_w}[n] = r[n]^* r[n]$ with size of $N_{null} + 1$
- (2) $\text{energy_w1}[n] = \sum_{m=n-N_{null}+1}^n \text{power_w}[m] = \text{energy_w1}[n-1] + \text{power_w}[n] - \text{power_w}[n-N_{null}]$
 $= \text{energy_w1}[n-1] + \text{power_w}[\text{end}] - \text{power_w}[1]$ with size of $N_{null} + 1$
- (3) $\text{sig_w}[n] = r[n]$ with size of $N_{fft} + 1$
- (4) $\text{corr_w}[n] = r[n]^* r[n-N_{fft}] = \text{sig_w}[\text{end}]^* \text{sig_w}[1]$ with size of $N_g + 1$
- (5) $\text{energy_w2}[n] = \sum_{m=n-N_g+1}^n \text{power_w}[m] = \text{energy_w2}[n-1] + \text{power_w}[n] - \text{power_w}[n-N_g]$
 with size of $N_{fft} + 1$

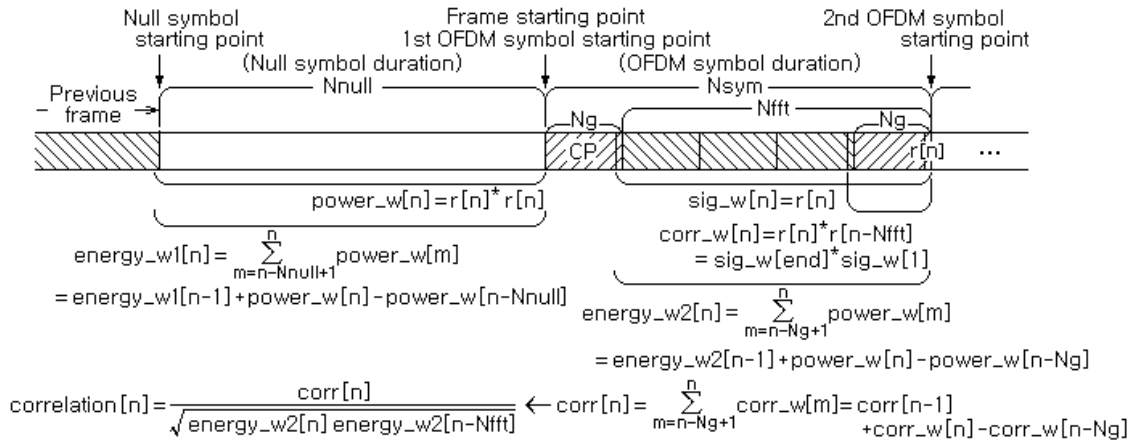


Figure 11.11 Window buffers maintained to catch the starting points of the null, frame, and OFDM symbols


```

%do_sync_w_double_window.m
% Copyleft: Won Y. Yang, wyyang53@hanmail.net, CAU for academic use only
clear, clf
Cor_thd=0.988; % The threshold to determine the peak of correlation
Nfft=64; Ng=16; Nsym=Nfft+Ng; Nsym1=Nsym+1;
Nnull=Nsym; Nw=Nnull; Nw1=Nw+1; Nw2=Nw*2; Ng1=Ng+1; Nfft1=Nfft+1;
Nd=90; % Remaining period of the last symbol in the previous frame
N_OFDM=3; % One Null + N_OFDM symbols
Max_energy_ratio=0; Min_energy_ratio=1e10;
r = [rand(1,Nd)-0.5+j*(rand(1,Nd)-0.5) zeros(1,Nnull)];
for i=1:N_OFDM
    symbol=rand(1,Nfft)-0.5 +j*(rand(1,Nfft)-0.5);
    r = [r symbol(end-Ng+1:end) symbol];
end
L_frame = length(r);
r = r + 0.1*(rand(1,L_frame)-0.5+j*(rand(1,L_frame)-0.5));
energy_w1=zeros(1,Nw1); power_w=zeros(1,Nw1);
sig_w=zeros(1,Nfft1); energy_w2=zeros(1,Nfft1); corr_w=zeros(1,Ng1);
OFDM_start_points = [0]; corr=0;
for n=1:L_frame
    sig_w = [sig_w(2:end) r(n)]; % Signal window
    power_n = r(n)'*r(n); % Current signal power
    power_w = [power_w(2:end) power_n]; % Power window
    energy_w1=[energy_w1(2:end) energy_w1(end)+power_n]; %Energy window
    if n>Nw, energy_w1(end)=energy_w1(end)-power_w(1); end %of size Nw
    energy_w2=[energy_w2(2:end) energy_w2(end)+power_n]; %Energy window
    if n>Ng, energy_w2(end)=energy_w2(end)-power_w(end-Ng); end
    corr_w(1:end-1) = corr_w(2:end);
    if n>Nfft
        %Correlation between signals at 2 points spaced Nfft samples apart
        corr_w(end)=abs(sig_w(end)'*sig_w(1)); corr=corr+corr_w(end);
    end
    if n>Nsym, corr=corr-corr_w(1); end %Correlation window of Ng pts
    % Null Symbol detection based on energy ratio
    if n>=Nw2
        energy_ratio = energy_w1(end)/energy_w1(1);
        energy_ratios(n) = energy_ratio;
        if energy_ratio<Min_energy_ratio % Eq.(11.3.8a)
            Min_energy_ratio = energy_ratio; Null_start_point = n-Nw+1;
        end
        if energy_ratio>Max_energy_ratio % Eq.(11.3.8b)
            Max_energy_ratio = energy_ratio; F_start = n-Nw+1;
        end
    end
end
% CP-based Symbol Time estimation
if n>Nsym
    % Normalized, windowed correlation across Nfft samples for Ng pts
    correlation=corr/sqrt(energy_w2(end)*energy_w2(1)); % Eq.(11.3.9)
    correlations(n) = correlation;
    if correlation>Cor_thd&n-Nsym>OFDM_start_points(end)+Nfft
        OFDM_start_points = [OFDM_start_points n-Nsym+1];
    end
end
end
end

```

```

Estimated_start_points=
[Null_start_point F_start OFDM_start_points(2:end)]
True_start_points=[Nd+1:Nsym:L_frame]
N_True_start_points=length(True_start_points);
subplot(311), stem(real(r)), set(gca,'XTick',True_start_points)
hold on, stem(True_start_points,0.9*ones(1,N_True_start_points),'k*')
N_Sym=length(Estimated_start_points);
stem(Estimated_start_points,1.1*ones(1,N_Sym),'rx')
title('Estimated Starting Points of Symbols')
subplot(312), semilogy(energy_ratios), set(gca,'XTick',True_start_points)
title('Ratio of 2 Successive Windowed Energies for Nw samples')
subplot(313), plot(correlations)
hold on, title('Correlation across Nfft samples')

```

At every instant when a sampled signal arrives, the normalized and windowed correlation value

$$\text{correlation}[n] = \frac{\sum_{m=n-N_g+1}^n \text{corr_w}[m]}{\sqrt{\text{energy_w2}[n]\text{energy_w2}[n-N_{fft}]}} \stackrel{?}{>} \text{Threshold} \quad (11.3.9)$$

is computed to determine if it is the correlation value between the CP and the last N_g samples of an OFDM symbol, i.e. if the current sample is the end of an OFDM symbol or not. If the normalized correlation value is found to exceed some threshold, say, 0.988 at n , the starting point of the detected OFDM symbol is determined to be $n - N_{sym} + 1$ (one OFDM symbol duration before the detection time) where $N_{sym} = N_g + N_{fft}$ is the *OFDM symbol duration*. Compared with this symbol timing process, catching the starting points of a null symbol and a frame based on the energy ratio between two successive blocks is very misty since there can be no normalization of energy ratio and accordingly, the appropriate threshold values to determine the maximum and minimum of the energy ratio are difficult to fix. Let us run the MATLAB program “do_sync_w_double_window.m” to get Fig. 11.12 together with the following result:

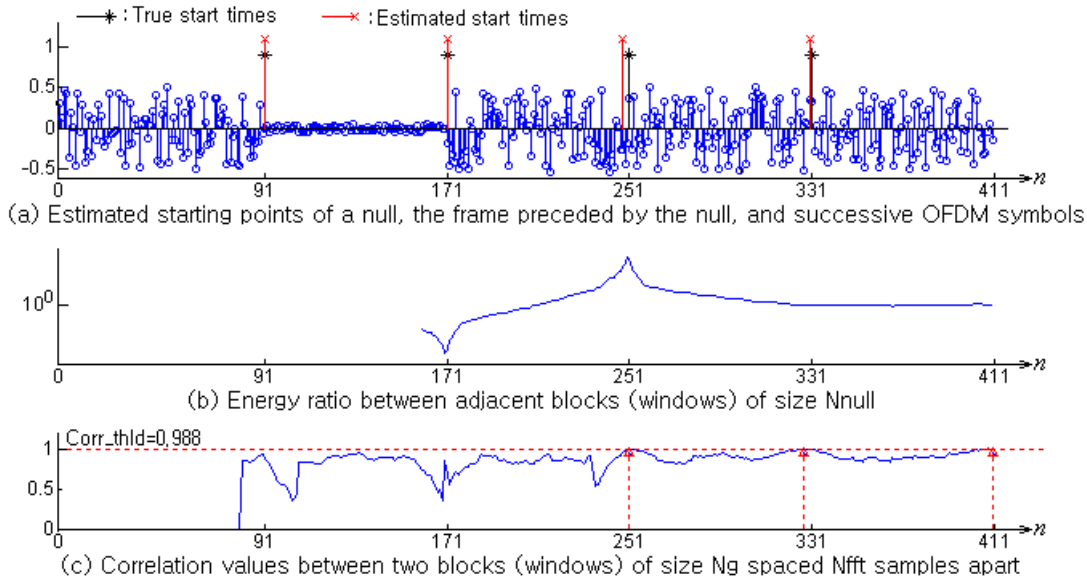


Figure 11.12 An example of simulation results obtained by running “do_sync_w_double_window.m”

```
>> do_sync_w_double_window
    Estimated_start_points =    91    171    171    248    330
    True_start_points     =    91    171    251    331
```

The detection of the starting points of a null, the frame preceded by the null symbol, and successive OFDM symbols seems to be successful. That is right as far as the OFDM symbol starting points are concerned. However, the maximum/minimum points of the energy ratio for estimating the starting points of a null/frame (shown in Fig. 11.12(b)) have not been detected on-line since they could be recognized only after a considerable time span. In this aspect, the above program has a room for improvement towards more practical detection of the starting points of a null and the frame preceded by the null symbol, possibly in concert with the OFDM symbol starting point detection.

2. Integral Frequency Offset Estimation

The *CFO* (carrier frequency offset) that may be caused by the mismatch of the oscillators in the transmitter and receiver can be regarded as the sum of the *IFO* (integral frequency offset) and the *FFO* (fractional frequency offset) where the IFO and the FFO are an integer and a fraction times the subcarrier frequency interval $\Omega_0 = 2\pi/N_{fft}$, respectively. The IFO can be estimated based on the correlation between the received PRS(phase reference symbol)-FFT signal $Y_{PRS}(k)$ and the local (frequency-domain) PRS signal $X_{PRS}(k)$ at the receiver by applying one of the following three methods:

APRS (Algorithm using PRS):

$$\Delta\hat{f}_i = \underset{d}{\text{ArgMax}} \left| \sum_{k=0}^{N_{fft}-1} Y_{PRS}(k) X_{PRS}^*(k-d) \right| \quad (11.3.10a)$$

ACIR (Algorithm using the channel impulse response):

$$\Delta\hat{f}_i = \underset{d}{\text{ArgMax}} \left\{ \underset{n}{\text{Max}} z[n] = \text{IFFT} \left\{ Y_{PRS}(k) X_{PRS}^*(k-d) \right\} \right\} \quad (11.3.10b)$$

AIDC (Algorithm using the intercarrier differential correlation):

$$\Delta\hat{f}_i = \underset{d}{\text{ArgMax}} \left| \sum_{k=0}^{N_{fft}-1} \underline{Y}_{PRS}(k) \underline{X}_{PRS}^*(k-d) \right| \quad (11.3.10c)$$

where

$$\underline{Y}(k) = Y(k)Y^*(k-1), \quad \underline{X}(k) = X(k)X^*(k-1) \quad (11.3.10d)$$

3. Fine Frame Synchronization

The *fine frame synchronization* can be performed by taking the peak point of the correlation between the received signal $y_{PRS}[n]$ and the time-domain PRS $x_{PRS}[n]$ where the (time-domain) correlation can equivalently be computed from the IFFT of the (frequency-domain) product of the received FFT signal $Y(k)$ and the IFO-compensated PRS $X_{PRS}^*(k - \Delta\hat{f}_i)$:

$$\hat{n}_f = \underset{n}{\text{ArgMax}} \left\{ z[n] = \text{IFFT} \left\{ Y(k) X_{PRS}^*(k - \Delta\hat{f}_i) \right\} \right\} \quad (11.3.11)$$

4. Fractional Frequency Offset Estimation

With the same idea as Eq. (11.3.2b), the fractional frequency offset (FFO) estimate can be obtained from $1/2\pi$ times the phase difference between the two (fine frame-synchronized) blocks of size N_w spaced N_{fft} samples apart:

$$\Delta \hat{f}_f = \frac{1}{2\pi} \angle \left\{ \sum_{n=1}^{N_w} y[n + \hat{\tau}_f + N_{fft}] y^*[n + \hat{\tau}_f] \right\} \quad (11.3.12)$$

where the two blocks are supposed to agree with each other without CFO and noise and the window size N_w is often set equal to the guard interval or cyclic prefix (CP) size N_g . Note that a large window size will help increasing the accuracy of the FFO estimation.

```
function IFO_est=IFO_estimate(y,X,IFO_range)
% To estimate the IFO (Integral Frequency Offset)
% y: A received time-domain signal, supposedly containing ifft(PRS)
% X: (frequency-domain) Phase Reference Symbol
% IFO_range : Range of possible IFOs to be searched
M=3; Max=zeros(1,M); IFO_est=zeros(1,M); % 3 methods to estimate IFO
Nfft=length(X); Y = fft(y,Nfft);
Ybar=Y.*conj(Y([Nfft 1:Nfft-1])); % Eq.(11.3.10)
Xbar=X.*conj(X([Nfft 1:Nfft-1]));
for i=1:length(IFO_range)
    d=IFO_range(i); YX = Y.*conj(rotate_r(X,d));
    Mag(1) = abs(sum(YX)); % Eq.(11.3.10a) APRS
    Mag(2) = max(abs(ifft(YX))); % Eq.(11.3.10b) ACIR
    Mag(3) = abs(Ybar*rotate_r(Xbar,d)'); % Eq.(11.3.10c) AIDC
    for m=1:M
        if Mag(m) >= Max(m), Max(m)=Mag(m); IFO_est(m)=d; end
    end
end

function PRS=phase_ref_symbol()
% Nfft=Nsd+Nvc=1536+512=2048;
Nfft=1536+512; %Nsd(# of data subcarriers)+Nvc(# of virtual carrier)
h= ...
[0 2 0 0 0 0 1 1 2 0 0 0 2 2 1 1 0 2 0 0 0 0 1 1 2 0 0 0 2 2 1 1;
 0 3 2 3 0 1 3 0 2 1 2 3 2 3 3 0 0 3 2 3 0 1 3 0 2 1 2 3 2 3 3 0;
 0 0 0 2 0 2 1 3 2 2 0 2 2 0 1 3 0 0 0 2 0 2 1 3 2 2 0 2 2 0 1 3;
 0 1 2 1 0 3 3 2 2 3 2 1 2 1 3 2 0 1 2 1 0 3 3 2 2 3 2 1 2 1 3 2;
0 2 0 0 0 0 1 1 2 0 0 0 2 2 1 1 0 2 0 0 0 0 1 1 2 0 0 0 2 2 1 1];
n=[1 2 0 1 3 2 2 3 2 1 2 3 1 2 3 3 2 2 2 1 1 3 1 2 ...
 3 1 1 1 2 2 1 0 2 2 3 3 0 2 1 3 3 3 3 0 3 0 1 1];
jpi2 = j*pi/2;
for p=1:12 %12*4*32=1536
    for i=1:4
        if p<=6, il=i; else il=6-i; end
        for k=1:32
            temp_seq((p-1)*128+(i-1)*32+k)=exp(jpi2*(h(il,k)+n((p-1)*4+i)));
        end
    end
end
PRS = [zeros(1,256) temp_seq(1:768) 0 temp_seq(769:end) zeros(1,255)];
```

```

%do_sync_for_DMB.m
clear, clf
Nfft=2048; Ng=504; Nnull=2656;
CFO = -1.7; phase = 0; % A pseudo Carrier Frequency/Phase Offset
nF = 1 % A pseudo Frame Time Offset (delay)
PRS = phase_ref_symbol;
prs = ifft(PRS,Nfft); % Time-domain PRS
tx = [prs(Nfft-Ng+1:Nfft) prs]; % Add Cyclic Prefix
L_tx=length(tx);
% Set up the (pseudo) CFO (pretended)
Nd = 100; % Delay tolerance
y = set_CFO(tx,CFO,phase,Nfft); A=0.05;
y = [A*(rand(1,Nd)-0.5+j*(rand(1,Nd)-0.5)) y];
y = [y A*(rand(1,Nd)-0.5+j*(rand(1,Nd)-0.5))];
if nF>0
    y=[A*(rand(1,nF)-0.5+j*(rand(1,nF)-0.5)) y(1:end-nF)]; %Delayed
elseif nF<0
    y=[y(1-nF:end) A*(rand(1,-nF)-0.5+j*(rand(1,-nF)-0.5))]; %Advanced
end
% IFO (Integral Frequency Offset) estimation
IFO_range = [-3:3];
y_ifft = y(Nd+Ng+[1:Nfft]);
IFO_est = IFO_estimate(y_ifft,PRS,IFO_range); % Eq.(11.3.10a,b,c)
%In order to realize how critical the IFO estimate is for FFS,
% activate the following statement even with a very short delay nF=1;
% IFO_est = zeros(1,3);
% FFS (fine frame synchronization)
Y = fft(y_ifft,Nfft);
YX = Y.*conj(rotate_r(PRS,IFO_est));
[Max,nF_] = max(abs(ifft(YX))); % Eq.(11.3.11)
if nF_>Nfft/2, nF_=nF_-Nfft; end % Periodicity of FFT/IFFT
nF_h = (nF_-1)
% FFO (Fractional Frequency Offset) estimation
nn = Nd+[1:Ng]; nn1 = Nfft + nn;
% To realize the importance of reflecting the FFS into FFO estimation,
% activate the following statement with a delay nF=2;
% nF_h = 0;
FFO_est = angle(y(nF_h+nn1)*y(nF_h+nn)')/(2*pi) % Eq.(11.3.12)
% FFO estimate without incorporating the FFS result
FFO_est0 = angle(y(nn1)*y(nn)')/(2*pi) % Eq.(11.3.12) without nF_h
% Overall CFO estimate
CFO_est = IFO_est + FFO_est;
fprintf('\n For CFO=%10.8f,\n', CFO);
form = ' CFO_estimate(%10.8f) = IFO(%10.8f)+FFO(%10.8f)\n';
for i=1:3
    fprintf(form, CFO_est(i), IFO_est(i), FFO_est);
end
% CFO compensated symbols
y_compensated = compensate_CFO(y,CFO_est(3),Nfft);
% Discrepancy between the CFO compensated symbols and original ones
discrepancy = norm(tx-y_compensated(Nd+[1:L_tx]))/L_tx

```

The above program “do_sync_for_DMB.m” simulates the synchronization scheme depicted in Fig. 11.10(a) where the routines ‘IFO_estimate()’ and ‘phase_ref_symbol()’ are used to compute the IFO estimates by Eqs. (11.3.10a~c) and to generate the frequency-domain PRS (phase reference symbol), respectively.

11.4 CHANNEL ESTIMATION AND EQUALIZATION

A frequency-domain training sequence $X(k) = X_R(k) + jX_I(k)$ with the corresponding channel output $Y(k) = Y_R(k) + jY_I(k)$ can be used for channel estimation as

$$\hat{H}(k) = \frac{Y(k)}{X(k)} = \frac{X^*(k)Y(k)}{X^*(k)X(k)} = \frac{(X_R(k)Y_R(k) + X_I(k)Y_I(k)) + j(X_R(k)Y_I(k) - X_I(k)Y_R(k))}{X_R^2(k) + X_I^2(k)} \quad (11.4.1)$$

where $X(k)$ and $Y(k)$ are the FFTs of the (long preamble) input and the corresponding output of the channel, respectively, $X_R(k)$ and $Y_R(k)$ are the real parts, and $X_I(k)$ and $Y_I(k)$ are the imaginary parts of $X(k)$ and $Y(k)$, respectively. Noting that in the IEEE Standard 802.11a, the long preamble used for channel estimation is $X(k) = X_R(k) + jX_I(k) = 1$ or -1 (with $X_I(k) = 0$) as shown in Fig. 11.6(b), the channel estimator (11.4.1) can be simplified as

$$\hat{H}(k) = \frac{Y(k)}{X(k)} = X_R(k)(Y_R(k) + jY_I(k)) = X(k)Y(k) \quad (11.4.2)$$

Since the long preamble contains two repeated training sequences, the average of the FFTs ($Y_1(k)$ and $Y_2(k)$) of the channel outputs to the two consecutive training sequences can be taken for better channel estimation:

$$\hat{H}(k) = \frac{1}{2} X(k)(Y_1(k) + Y_2(k)) \quad (11.4.3)$$

This estimation scheme is cast into the MATLAB routine ‘channel_estimate()’. The following program “do_channel_estimation.m” uses this routine to estimate a channel based on the frequency-domain long preamble $X(k)$ and the FFT $Y(k)$ of the output of the channel to the (time-domain) long preamble. It also uses another routine ‘equalizer_in_freq()’ to equalize the output to the unknown input by compensating the channel effect:

$$\hat{X}(k) = \frac{Y(k)}{\hat{H}(k)} \quad (11.4.4)$$

```
function H_est=channel_estimate(X,y)
% X = Known frequency-domain training symbol
% y = Time-domain output of the channel
% H_est = Estimate of the channel (frequency) response
if length(X)>52, X = X([1:26 28:53]); end % for k=[-26:-1 1:26]
y1 = y(32+[1:64]); y2 = y(96+[1:64]);
Y1 = fft(y1); Y1 = Y1([39:64 2:27]); % Arranged in the -/+ frequency
Y2 = fft(y2); Y2 = Y2([39:64 2:27]); % Arranged in the -/+ frequency
H_est = X.*(Y1+Y2)/2; % Eq. (11.4.3)
```

```

%do_channel_estimation.m
clear, clf
% Read the time-domain channel response stored in a 64x2 matrix
load ch_complex.dat;
h=(ch_complex(:,1)+j*ch_complex(:,2)).'; % Time-domain channel response
Nfft=64; Ng=16; Nsym=Nfft+Ng; Nnull=Nsym; Nw=2*Ng; Tsym=4e-6;
A_sig=0.5; A_noise=0.01; % Amplitudes of signal and noise
Nd=120; % Remaining period of the last symbol in the previous frame
[s_preamble,Short] = short_train_seq(Nfft);
[l_preamble,Long] = long_train_seq(Nfft);
% Make a pseudo received sequence
t_frame=[A_sig*(rand(1,Nd)-0.5) zeros(1,Nnull) s_preamble l_preamble];
symbol=A_sig*(rand(1,Nd)-0.5); symbol=[symbol(end-Ng+1:end) symbol];
t_frame = [t_frame symbol]; L_frame = length(t_frame);
noise = A_noise*(rand(1,L_frame)-0.5 +j*(rand(1,L_frame)-0.5));
r_frame = channel(t_frame,h) + noise;
True_STO = Nd+Nnull+Nsym*2+1 % Starting point of the long preamble
STO = True_STO
% STO estimation is critical to the performance of channel estimation
% To realize this, change the above line into STO=True_STO+1 or -1
H_est = channel_estimate(Long,r_frame(STO+[0:159]));
% The true frequency-domain channel response is obtained from the FFT
% of the time-domain channel (impulse) response.
H = fft(h,Nfft); % k=0(1):26(27) 27(28):37(38) 38(39):63(64)
H_true = H([39:64 2:27]); % Arranged in -/+ frequency k=-26:-1 1:26
discrepancy_H_est_and_H_true = norm(H_est-H_true)/norm(H_true)
% Let's see how the channel equalizer with the estimated channel
% response (H_est) works.
X = rand(1,52)-0.5+j*(rand(1,52)-0.5); % for k=[-26:-1 1:26]
X_arranged = [0 X(27:end) zeros(1,11) X(1:26)]; % in +/- frequency
x = ifft(X_arranged); x_CP = [x(49:64) x]; % IFFT and add CP
y = channel(x_CP,h); % Channel output to an arbitrary input with CP
Y = fft(y(17:80)); % Remove CP and FFT
Y=Y([39:64 2:27]); % Arranged in -/+ frequency for k=[-26:-1 1:26]
Yeq = equalizer_in_freq(Y,H_est);
discrepancy_X_and_Y = norm(X-Y)/norm(X) % With no channel compensation
discrepancy_X_and_Yeq = norm(X-Yeq)/norm(X) % With channel equalizer

function [y,ch_buf] = channel(x,h,ch_buf)
L_x = length(x); L_h = length(h); h=h(:);
if nargin<3, ch_buf = zeros(1,L_h);
else L_ch_buf = length(ch_buf);
if L_ch_buf<L_h, ch_buf = [ch_buf zeros(1,L_h-L_h)];
else ch_buf = ch_buf(1:L_h);
end
end
for n=1:L_x, ch_buf=[x(n) ch_buf(1:end-1)]; y(n)=ch_buf*h; end

function Y_eq = equalizer_in_freq(Y,H_est)
H_est(find(abs(H_est)<1e-6))=1; Y_eq = Y./H_est; % Eq.(11.4.4)

% ch_complex.dat
0.4923667628304478349754 -0.3721824742433228472294
-0.5175114648028624753096 -0.3043313718301440817804
.....

```

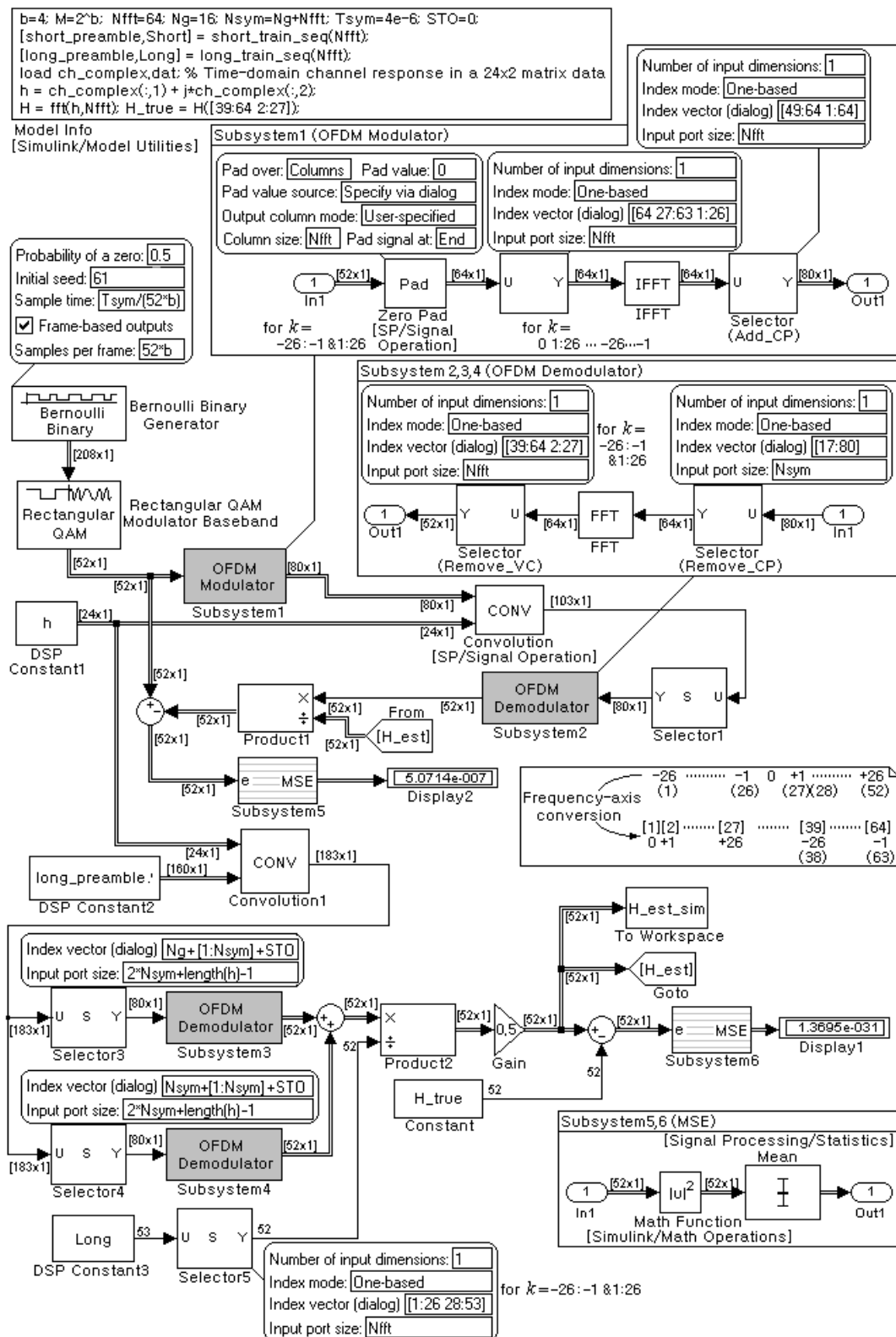


Figure 11.13 Simulink model for channel estimation and compensation ("channel_estimation_sim.mdl")

INDEX

A

adaptive equalizer (ADE), 154-155
 address decoding circuit, 277
 Alamouti, 313
 A-law, 92
 analog frequency, 20
 analytic signal, 30, 33
 angle modulation, 82
 antipodal (bipolar) signaling, 114, 122
 ASK (amplitude shift keying), 169
 asynchronous, 78
 autocorrelation, 29, 30, 50, 55, 58
 autocorrelation function, 50
 autocovariance function, 50
 average signal energy of QAM symbol, 196
 AWGN (additive white Gaussian noise), 108, 136

B

balance property (of PN), 340
 bandpass Gaussian noise, 49, 62
 bandwidth efficiency, 200, 203, 268
 Bayes' rule, 40
 BCJR (Bahl-Cocke-Jelinek-Raviv), 301
 BCH (Bose-Chaudhuri-Hocquenghem) code, 284, 322
 belief propagation algorithm, *see* BPA
 Bessel function, 24, 26, 50
 BFSK (binary phase shift keying) signal, 22
 binary symmetric channel (BSC), 264, 321
 bi-orthogonal signaling, 133-134
 bit error probability, 129, 131
 bit error rate (BER), *see* bit error probability
 block coding, 271, 284
 Blockset, 423
 Communication ~, 424
 Signal Processing ~, 423
 BPA (belief propagation algorithm), 310
 branch cost (metric), 289
 BSC, *see* binary symmetric channel

C

capacity boundary, 268
 carrier frequency offset (CFO), 362-363, 376
 carrier phase recovery, 225, 233, 248-252
 ~ for BPSK, 248
 ~ for PSK, 251-252
 ~ for QAM, 238-242, 250

 ~ for QPSK, 249
 carrier recovery, 225, 236, 239-240
 carrier timing recovery, 252-253
 Carson's (bandwidth) rule, 36, 83
 CDMA (code division multiple access), 354
 central limit theorem (CLT), 47, 48
 centroid, 87
 CFO (carrier frequency offset), 362-363, 376
 ~ estimation, 365-369, 398-405
 channel capacity, 265-269, 319, 321
 channel coding, 263, 269
 channel estimation, 379, 381, 405
 channel reliability, 300, 309
 Chebyshev inequality, 43
 check node (c-node), 310-313
 chip interval, 347
 circular correlation, 69
 coarse CFO estimate, 365-366, 404
 coarse frame synchronization, 373
 coarse frequency offset (CFO), 362-363, 376
 code division multiple access (CDMA), 354
 code efficiency, 257
 code rate, 265, 285
 code vector, 273, 309
 codeword, 257, 265, 270-276, 279-281, 287
 codeword matrix, 270, 273
 coding gain, 317-318
 co-error function, *see* complementary error function
 coherent, 75, 82, 171, 180-183, 190, 206
 colored noise, 53
 Communication Blockset, 424
 Communication Toolbox, 284
 compansion, 93
 complementary error (co-error) function, 43, 118
 complex envelope, 33
 conditional entropy, 264
 conditional expectation, 47
 conditional probability, 40
 conditional probability density function, 41
 conjugate symmetry, 13
 constraint length, 285, 286
 constraint length vector, 292, 293, 325, 326, 328
 controlled ISI signaling, 143
 conventional AM, 75
 convolution, 14
 convolutional code, 285, 287
 convolutional coding, 285
 convolutional encoder, 285, 289, 325

correlation, 14, 29, 44, 375
 circular \sim , 69
 correlation coefficient, 44
 correlation property (of PN), 340
 correlator, 112-113
 Costas loop, 236-237, 244, 249
 covariance, 44
 covariance matrix, 45
 CP (cyclic prefix), 358, 365, 371, 377, 393
 crosscorrelation, 50, 52, 58, 111, 270
 crosscorrelation function, 50
 crosscovariance, 45
 crosscovariance function, 50
 crossover probability, 271, 272, 277
 CTFS (continuous-time Fourier series), 1, 407, 408
 CTFT (continuous-time Fourier transform),
 7, 27, 407, 409-411
 cycle, 310
 cyclic code, 280, 284
 cyclic decoder, 282, 283
 cyclic encoder, 282, 283
 cyclic prefix (CP), 358, 365, 371, 377, 393

D

debugging (MATLAB), 419
 decision-feedback equalizer (DFE), 155-158
 deinterleaving, 382
 delta modulation (DM), 100-101
 delta-sigma modulation, 105
 depuncturing, 384
 despreading, 346, 347
 deviation, 43
 DFE (decision-feedback equalizer), 155-158
 DFS (discrete Fourier series), 19
 DFT (discrete Fourier transform), 19
 differential PSK (DPSK), 190-195
 differential PCM (DPCM), 97-99
 differential space-time block code, *see* DSTBC
 differentiation w.r.t. a vector, 414
 digital frequency, 20
 discrete memoryless channel (DMC), 263
 DM (delta modulation), 100-101
 DMC (discrete memoryless channel), 264
 Doppler effect, 61
 DPCM (differential pulse code modulation), 97-99
 DS(direct sequence)-SS, 345-349
 DSB(double sideband)-AM, 71
 DSB-AMSC, 71-72
 DSM (delta-sigma modulation), 105
 DSTBC (differential space-time block code),
 314, 333
 DTFS (discrete-time Fourier series), 19
 DTFT (discrete-time Fourier transform), 18, 27
 duality, 11, 17

duobinary precoding, 146
 duobinary signaling, 143-144, 146-147, 165-167

E

early-late gate timing recovery, 241-243
 energy-type signal, 29
 entropy, 256, 319-320
 envelope detector, 76, 173, 182
 equalizer, 148-158, 167
 adaptive \sim , 154
 decision-feedback \sim , 155
 minimum mean-square error (MMSE) \sim , 151
 zero-forcing \sim (ZFE), 148
 equalization 379
 ergodic, 52
 error correcting capability, 272, 284, 322
 error detecting capability, 272
 error floor, 308
 error function, 43
 error pattern, 274-275, 281-282, 322-323
 error probability, 122, 123, 201, 202
 \sim for antipodal signaling, 117
 \sim for BASK with coherent detection, 178
 \sim for BASK with non-coherent detection, 174
 \sim for bi-orthogonal signaling, 134-135
 \sim for BPSK, 187
 \sim for DPSK (differential PSK), 193
 \sim for FSK with coherent detection, 179
 \sim for FSK with non-coherent detection, 182-183
 \sim for multidimensional signaling, 130-131
 \sim for multi-level signaling, 128-129
 \sim for OOK signaling, 118
 \sim for orthogonal signaling, 121
 \sim for PAM, 171
 \sim for PSK, 188
 \sim for QAM, 197
 ESD (energy spectral density), 29
 EUREKA-147 DAB, 372
 expectation, 43
 extended Golay code, 279
 extrinsic information, 308

F

fading, 60
 \sim amplitude, 301
 fast \sim , 61
 (frequency-)flat \sim , 61
 (frequency-)selective \sim , 61
 Raleigh \sim , 60
 Rician \sim , 60
 slow \sim , 61
 fast FH-SS, 350-352
 feedback shift register, 280, 281, 283, 337-341

FFO (fractional frequency offset), 376, 377
 FH(frequency hopping)-SS, 350-353
 fast ~, 350-352
 slow ~, 350-351, 355
 filtering, 57
 fine CFO estimate, 365-366, 405
 fine frame synchronization, 376
 finite pulsewidth sampler, 15
 flat fading, 61
 FM (frequency modulation), 82-83
 fractional frequency offset (FFO), 376, 377
 frame synchronization, 373, 376
 free distance, 289
 frequency aliasing, 28
 frequency deviation ratio, 36
 frequency modulation (FM), 82-83
 frequency offset, 376-377
 frequency response, 7, 14, 18, 58, 63
 ~ of a discrete-time LTI system, 63
 frequency-selective fading, 61
 frequency shifting, 6, 14
 FSK (frequency shift keying), 170, 178-186
 function of a random variable, 42
 fundamental frequency, 20

G

Gallager, 309
 Gaussian, 43-46, 49, 205, 235
 Gaussian noise, 49, 205
 sampled ~, 57
 generator matrix, 272-274, 278, 285, 325, 327
 generator polynomial, 280, 285
 generator sequence, 285
 Golay code, 279
 Gold code, 340-343
 Gold sequence, *see* Gold code
 gradient, 154, 414
 granular (idling) noise, 100-101
 guard interval, 358, 362, 365, 387, 398

H

Hadamard matrix, 270
 Hamming code, 278, 284, 321
 Hamming distance, 272
 Hamming weight, 272, 287
 hard value, 298
 hard decision, 289, 292
 Hermitian symmetry, 13
 Hilbert transform, 32, 37, 78
 Huffman code, 257-259, 320

I

ideal LPF (lowpass filter), 140

ideal sampler, 15
 IEEE Standard 802.11a, 386-388
 timing-related parameters, 387
 IFO (integral frequency offset), 376
 impulse function, 5
 impulse sequence, 6
 impulse train, 5
 independent, 41-42, 45
 inner (or dot or scalar) product, 121
 in-phase component, 33, 49
 instant sampler, 15
 integral frequency offset (IFO), 376
 interleaving, 382
 ISI (intersymbol interference), 139
 ISI free condition, 141-142

J

Jacobian, 42, 43, 49, 412
 jamming 347
 jointly normal distribution, 44
 joint probability, 40
 joint probability density function, 41

K

Kasami sequence, 341-342

L

L-value, 298, 305
 Laplace transform, 412
 LAPP (log a posteriori probability), 301
 LDPC (low-density parity-check)
 ~ code, 284, 309
 ~ decoding, 310
 ~ encoding, 309
 least mean square (LMS), 154
 Lempel-Ziv-Welch (LZW) coding, 260
 Lempel-Ziv-Welch (LZW) decoding, 261-262
 LF (loop filter), 226-230
 LFSR, *see* linear feedback shift register
 line code, 65
 linear block coding, 271, 284
 linear feedback shift register (LFSR),
 281, 283, 337-338, 340-341
 Lloyd-Max, 89-90
 LLR (log-likelihood ratio), 298, 301, 305
 conditioned ~, 301, 308, 310
 LMS (least mean square), 154
 log-MAP decoding, 298
 long preamble, 363-365
 long training sequence, *see* long preamble
 loop filter (LF), 226-230
 lowpass equivalent, 33
 LSSB (lower single sideband), 80

LZW (Lempel-Ziv-Welch) coding, 260
 LZW (Lempel-Ziv-Welch) decoding, 262

M

m -sequence (maximal length sequence), 339
 ~ generator, 339
 preferred pairs of ~s, 340
 MAP (maximum a posteriori probability), 298, 301
 marginal probability density function, 41
 Mason's gain formula, 287
 matched filter, 110-111
 MATLAB Command Window, 418
 MATLAB Editor Window, 418
 MATLAB introduction, 417
 maximal length sequence (m -sequence), 339
 maximum a posteriori probability (MAP), 298
 maximum likelihood estimate (MLE), 231
 mean, 43, 55
 mean function, 50
 mean square quantization error (MSQE), 87
 message passing algorithm (MPA), 310
 minimum distance, 123, 271, 272, 289
 minimum mean-square error equalizer, *see* MMSEE
 minimum-shift keying (MSK), 211-215
 minimum tone spacing, 179, 181
 MLE (maximum likelihood estimate), 231
 MMSEE (minimum mean-square error equalizer),
 151, 153
 modified duobinary signaling, 147-148
 modulation, 6, 14, 71
 modulation order, 127, 204
 modulation efficiency, 76
 MPA (message passing algorithm), 310
 MSK (minimum-shift keying), 211-215
 MSQE (mean square quantization error), 87
 μ -law, 92
 multi-amplitude, 127, 132
 multi-dimensional (orthogonal), 129, 132
 multi-level (multi-amplitude), 127, 132
 multi-path channel, 59
 mutual information, 265

N

NDA-ELD (nondata-aided early-late delay),
 244-246
 node cost (metric), 289
 noise power (or variance), 123
 noise PSD, 55
 noncoherent, 78, 171-173, 181-183, 190, 193, 206
 nondata-aided early-late delay (NDA-ELD),
 244-246
 nonuniform quantization, 89-91, 94
 normal convergence theorem, 47

(*see also* central limit theorem)
 normal distribution, 43, 44
 Nyquist band, 143, 144
 Nyquist bandwidth constraint, 140
 Nyquist frequency, 28

O

OFDM, 357-358
 OFDM receiver, 392
 OFDM symbol, 358, 362, 388, 394
 OFDM symbol duration, 371, 375, 388
 OFDM transmitter, 392
 offset QPSK (OQPSK), 207-208
 on-off keying (OOK), 118, 122
 OOK (on-off keying), 118
 OQPSK (offset QPSK), 207-208
 orthogonal, 45, 119, 121, 122, 129, 179
 orthogonal signaling, 119-121, 126, 132
 orthogonality, 397, 398

P

PAM (pulse amplitude modulation), 15, 17, 127
 parity check matrix, 274, 278, 279, 309
 Parseval's relation, 18
 partial response signaling, 143
 path metric, 305, 308
 PCM (pulse code modulation), 95-97
 perfect (code), 279
 phase-locked loop, *see* PLL
 phase modulation (PM), 82-83
 phase offset, 73
 phase shift keying (PSK), 170, 187-190
 phase tracker, 403
 physically realizable, 141
 $\pi/4$ -shifted QPSK (quadrature PSK), 209-210
 pilot, 365, 387, 388
 pilot polarity sequence, 387-388
 PLCP (physical layer convergence procedure),
 386
 PLL (phase-locked loop), 226-232, 247
 PM (phase modulation), 82-83
 PN (pseudo or pseudo-random noise), 337-338
 posteriori probability, 40, 231
 power efficiency, 200, 203
 power spectral density (PSD), 29, 53, 57-58
 power theorem, 18
 power-type signal, 29
 PPDU (PHY packet data unit), 386
 PRBS (pseudo-random binary sequence), 337
 preamble, 365
 precoding, 146
 predictor, 98
 pre-envelope signal, 30

prefix, *see* CP
 principal frequency range, 21
 priori information, 298, 307, 308, 310
 priori probability, 40
 probability, 39
 probability density function, 41
 probability distribution function, 41
 probability transition matrix, 264
 processing gain, 347, 350
 PSD (power spectral density), 29, 53, 57-58
 pseudo noise (PN), 337-338, 387
 PSK (phase shift keying), 170, 187-190
 pulse amplitude modulation (PAM), 15, 17, 127
 pulse code modulation (PCM), 95-97
 puncturing, 298, 384

Q

QAM (quadrature amplitude modulation), 195-200
 QPSK (quadrature PSK), 189-190
 $\pi/4$ -shifted ~, 209-210
 offset ~ (OQPSK), 207-208
 staggered ~, *see* OQPSK
 quadrature amplitude modulation (QAM), 195-200
 quadrature carrier, 197
 quadrature component, 33, 49
 quadrature correlator, 197
 quantization, 87-91, 94

R

raised-cosine (RC) filter, 160-164
 raised-cosine frequency response, 141-142, 159
 raised-cosine impulse response, 142, 160
 random process, 49
 Rayleigh fading, 60
 Rayleigh probability density function, 50, 174, 182
 RC filter, 109-110
 real convolution, 14
 rectangular pulse, 8
 rectangular wave, 2
 recursive systematic convolutional encoder, 298
 Reed-Solomon (RS) code, 284
 resolution frequency, 20
 Rice probability density function, 50, 62, 174, 182
 Rician fading, 60
 roll-off factor, 141
 RSC encoder, 298
 run property (of PN), 340

S

sampling frequency, 27
 sampling theorem, 27, 28
 scrambler, 387, 388
 Shannon-Hartley channel capacity theorem, 267

Shannon limit, 268
 short preamble, 363-365
 short training sequence, *see* short preamble
 sigma-delta modulation, *see* delta-sigma modulation
 signal constellation diagram, 122
 signal correlator, 112
 Signal Processing Blockset, 423
 signal space, 121, 122
 signal-to-quantization noise ratio (SQNR), 87
 Simulink Library Browser Window, 422
 sinusoidal FM (frequency-modulation) signal, 24
 slope overload distortion, 100-101
 slow fading, 61
 slow FH-SS, 355
 soft value, 297
 soft-decision, 289, 293
 soft-in/soft-output Viterbi algorithm, *see* SOVA
 source coding, 257, 263
 SOVA (soft-in/soft-output Viterbi algorithm), 305
 SPA (sum-product algorithm), 310, 311
 space-time block code (STBC), 313
 spread-spectrum (SS), 337
 spreading, 346, 347
 SQNR (signal-to-quantization noise ratio), 87
 square-root raised-cosine (SRRC) filter, 162-164
 square-root raised-cosine impulse response, 163
 squaring loop, 233-235, 248
 squaring timing recovery, 252
 SS (spread-spectrum), 337
 DS(direct sequence)-~, 345-349
 FH(frequency hopping)-~, 350-353, 355
 SSB(single-sideband)-AM, 78
 staggered QPSK, *see* OQPSK
 state diagram for convolution encoder, 288
 stationary, 51-52
 STBC (space-time block code), 313
 steepest descent method, 154
 STO (symbol time offset), 362-363
 ~ estimation, 370-372, 382, 393, 399
 stochastic process, 49
 strict-sense stationary, 51
 subcarrier, 358
 sum-product algorithm, *see* SPA
 suppressed carrier, 71
 symbol error probability, 128, 130, 131
 symbol error rate (SER),
 see symbol error probability
 symbol synchronization, 225, 241
 symbol time offset (STO), 362-363
 synchronous, 75, 82
 synchronization, 241, 365, 372, 393
 syndrome, 274-277, 281-282, 309, 322
 systematic, 274, 298

T

Tanner graph, 310
 TCM, 294-297, 329-334
 time shifting, 6, 14
 timing recovery, 241-245
 traceback depth, 292, 293
 transmission bandwidth, 75, 80
 transmission mode, 372
 transmitted carrier, 75
 trellis, 288-290, 293, 304
 trellis-coded modulation, *see* TCM
 triangular pulse, 8
 triangular wave, 2
 turbo code, 298-308
 turbo decoding, *see* log-MAP or SOVA

U

uncorrelated, 44, 45
 Ungerboeck TCM encoder, 329-330
 uniform distribution, 43, 46
 uniform number, 46
 uniform quantization, 88
 unit signal waveform, 108

USSB (upper single sideband), 78

V

variable node (v-node), 310, 311, 312
 variance, 43, 55
 VCO, 226-232, 247
 vector quantization, 103-104
 Viterbi
 ~ (decoding) algorithm, 289-293
 ~ decoding, 325
 voltage-controlled oscillator, *see* VCO

W

waveform coding, 270
 weight, 271
 white noise, 53, 55, 62, 108, 136, 205
 wide-sense stationary, 51

Z

z -transform, 412-413
 zero-forcing equalizer (ZFE), 148, 151

Index for MATLAB Routines (*: MATLAB built-in function)

MATLAB routine name	Description	Page number
ADC()	Analog-to-digital conversion	93
ade()	tunes the adaptive equalizer	154
Alaw()	A -law (Eq. (4.1.8a))	92
Alaw_inv()	A^{-1} -law (Eq. (4.1.8b))	92
awgn()*	Additive white Gaussian noise	136
awgn_()	Additive white Gaussian noise	136
bchgenpoly()*	makes the BCH code generator polynomial for given (N,K)	321
berawgn()*	Probability of bit error for various modulation schemes	202
bercoding()*	Probability of bit error for various coding schemes	279
bilinear()*	bilinear transformation (optionally with prewarping)	228-229
channel()	simulates the channel effect using convolution	380
channel_estimate()	finds the estimate of channel (frequency) response	379
coarse_CFO_estimate()	finds the coarse estimate of CFO	367
combis()	makes an error pattern matrix	275
compensate_CFO()	compensate the CFO	368
compensate_phase()	compensate the carrier phase	365
conv_encoder()	convolutional encoding	286
convenc()*	convolutional encoding	293, 325
corr_circular()	Circular (or cyclic or periodic) correlation	69, 343
CTFS()	CTFS coefficients together with the reconstruction	4, 5
CTFT()	CTFT spectrum together with the inverse CTFT	9
cyclic_decoder()	Cyclic decoder	282
cyclic_encoder()	Cyclic encoder	282
cyclpoly()*	makes the cyclic code generator polynomial for given (N,K)	281, 284

MATLAB routine name	Description	Page number
dc01e01	plots the CTFS spectra of rectangular/triangular waves	4
dc01e03	plots the CTFT spectra of rectangular/triangular pulses	9
dc01e16	plots the FFT spectrum of a FSK signal	23
dc01e17	plots the spectrum of a sinusoidal-modulated FM signal	25
dc01p02	Lowpass equivalent of a bandpass signal	38
dc0109_1	plots the Hilbert transform of a sinusoidal wave	31
dc0109_2	Lowpass equivalent of a bandpass signal	35
dc02e02a	plots the distribution of a uniform noise	46
dc02e02b	plots the distribution of a Gaussian noise	47
dc02e03	checks the validity of central limit theorem (CLT)	48
de02e05	plots the autocorrelation and histogram of a white noise	56
dc02f05	plots a white noise together with its correlation and PSD	54
dc02p01	plots the distribution of a bandpass noise with Rice pdf	62
dc04e03	Nonuniform quantization considering relative errors	93
dc0501	simulates binary communications with <i>RC</i> /matched filters	113
dc05f17	plots the theoretical BER curve for orthogonal signaling	131
dc07p01	simulates a linear combination of Gaussian noises	205
dc07t02	Table 7.2 (SNRdB for each signaling to achieve $BER=10^{-5}$)	203
dc09e05	constructs the codeword matrix for a linear block code	273
dc09p07	shows some examples of using convenc() and vitdec()	324
deci2bin1()	Decimal-to-binary conversion	272
decode()*	Decoding	284
decoder()	Decoding of a signal value by the given code table	96
deinterleaving()	Deinterleaving	383
dem_PSK_or_QAM()	performs the PSK or QAM demodulation	390
depuncture()	Depuncturing	384
detector_FSK()	Detection of FSK signals	217
detector_MSK()	Detection of MSK signals	222
detector_PSK()	Detection of PSK signals	220
dfe()	tunes the decision feedback equalizer (DFE)	156
do_ade	simulates the adaptive-equalizer (ADE)	155
do_BCH_BPSK_sim	runs the Simulink model "BCH_BPSK_sim"	323
do_CFO	simulates the CFO estimation and compensation	367
do_CFO_PHO_STO	simulates the CFO/PHO/STO estimation and compensation	399-401
do_channel_estimation	tries using channel_estimate() for channel estimation	380
do_cyclic_code	tries with a cyclic code	281
do_cyclic_codes	tries with cyclic codes	323
do_dfe	simulates the decision feedback equalizer (DFE)	156
do_FSK_sim	runs the Simulink model "FSK_passband_sim"	217
do_Hamming_code74	finds the BER performance of the Hamming (7,4) code	276
do_interleaving	tests the MATLAB routines interleaving() and deinterleaving()	383
do_mmsee	simulates the minimum mean-square error equalizer	152
do_MSK_sim	runs the Simulink model "MSK_passband_sim"	222
do_OFDM0	simulates a basic OFDM system	359
do_OFDM1	simulates an OFDM system (IEEE Std 802.11a)	389
do_PLL	simulates a PLL (phase-locked loop) system	229
do_PNG	uses MATLAB program 'png()' and runs Simulink model "PN_generator_sim.mdl" to generate a PN sequence	343

MATLAB routine name	Description	Page number
do_PSK_sim	runs the Simulink model “PSK_passband_sim”	220
do_puncture	tests the MATLAB functions puncture() and depuncture()	385
do_QAM_carrier _recovery	simulates QAM with decision-feedback carrier recovery	239
do_QAM_sim	runs the Simulink model “QAM_passband_sim”	222
do_QPSK_Costas	simulates QPSK using Costas loop for carrier phase recovery	237
do_QPSK_Costas_ earlylate	simulates QPSK using Costas loop and Early-Late algorithm for carrier phase recovery and timing recovery, respectively	244-245
do_rcos1	draws the impulse/frequency responses of a raised cosine filter	161
do_rcos2	performs the two cascaded square-root raised-cosine filtering	162
do_square_filter_clock	Square-law bit synchronizer	68
do_squaring_loop	simulates a squaring loop to get a reference signal from BPSK	234
do_STO	detects the OFDM symbol start points	393-395
do_STO_estimation	simulates the STO estimation/compensation	371
do_sym_sync_earlylate	simulates the early late sampling time control for symbol sync	243
do_sync_for_DMB	simulates the frame synchronization and CFO estimation	378
do_sync_w_double _window	simulates the estimation of OFDM frame and symbol start times using the double sliding window	374
do_TCM_8PSK	runs MATLAB routine ‘TCM’ and Simulink model	330
do_vector_quantization	simulates the vector quantization	103
do_vitdecoder	tries using vit_decoder() and vitdec()	290
do_vitdecoder1	tries the various usages of vitdec()	292
do_Viterbi_QAM	runs MATLAB routine ‘Viterbi_QAM.m’ together with Simulink model ‘Viterbi_QAM_sim.mdl’	327
do_zfe	simulates the zero forcing equalizer (ZFE)	150
DS_SS	simulates BPSK with DS-SS(spread spectrum)	348
encode()*	Encoding	284
equalizer_in_freq()	simulates the frequency-domain equalizer	380
FH_SS	simulates BFSK with fast FH-SS(spread spectrum)	352
FH_SS2	simulates BFSK with slow FH-SS(spread spectrum)	355
fine_CFO_estimate()	finds the fine estimate of CFO	367
freqz()*	Frequency response of a discrete-time system	63
Gauss_Hermite()	Integration using the Gauss-Hermite quadrature	132
GI_and_orthogonality	tests the orthogonality of an OFDM signal	398
gm2gM()	PNG polynomial into one used by Simulink	344
gray_code()	Gray coding	327
Hamm_gen()	makes the parity-check/generator matrices for Hamming code	278
Hammgen()*	makes the parity-check/generator matrices for Hamming code	278
hilbert()*	analytic signal with Hilbert transform as the imaginary part	76-77, 83
hist()*	plots a histogram	46-48, 56
Huffman_code()	Huffman code generator for a given symbol probability vector	258
IFO_estimate()	finds the estimate of IFO (integral frequency offset)	377
interleaving()	Interleaving	382
Jkb()	The first kind of kth-order Bessel function	25
LDPC_decoder()	Low-density parity-check (LDPC) decoder	311
LDPC_demo	simulates an LDPC coding and decoding	312
logmap()	Log-MAP algorithm for turbo coding	302
long_train_seq()	generates the long training sequence for 802.11a	364

MATLAB routine name	Description	Page number
LZW_coding()	performs the LZW coding on an input symbol sequence	260
LZW_decoding()	performs the LZW decoding on an input coded sequence	262
mmsee()	tunes the minimum mean-square error equalizer (MMSEE)	152
mod_PSK_or_QAM()	performs the PSK or QAM modulation	390
mulaw()	μ -law (Eq. (4.1.7a))	92
mulaw_inv()	μ^{-1} -law (Eq. (4.1.7b))	92
nextpow2 ()*	nextpow2(N) returns the first P such that $2^P \geq \text{abs}(N)$	74, 77, 82
OFDM_parameters()	set the OFDM parameters for IEEE Std 802.11a	388
phase_from_pilot()	finds the OFDM carrier phase estimate based on the pilot	365
phase_ref_symbol()	generates the PRS for EUREKA-147 DAB	377
plot_ds_ss	plots the signals in a DS-SS (spread spectrum) system	345
plot_MOD()	plots the signals appearing in a specified modulation process	74
PNG()	Pseudo-noise generator	338
poly2trellis()*	makes a trellis structure for a given generator polynomial	292-293
prdctr()	One-step-ahead predictor based on RLSE	99
prob_err_msg_bit()	Theoretical message bit error probability by Eq. (9.4.11)	276
prob_error()	Probability of error for various signaling	202
PSK_slicer()	slices each of PSK signals into the regular constellation points	316
puncture()	Puncturing	385
Q()	Complementary error (co-error) function	125
QAM()	QAM (quadrature amplitude modulation) modulation	327
QAM_dem()	QAM demodulation	328
quantize_nonuniform	Lloyd-Max nonuniform quantization	90
quantize_uniform	Uniform quantization	89
rand()*	generates an array of uniformly-distributed random numbers	47
randerr()*	generates a binary matrix having 0 or 1 at random positions	281, 284
randint()*	generates a matrix consisting of random integers	284
randn()*	generates an array of Gaussian-distributed random numbers	47
rcosflt()*	designs and implements a raised-cosine filter	160-163
rcosine()*	designs a raised-cosine filter	161-163
rD()	Rectangular pulse	9
rD_wave()	Rectangular wave	4
Rice_pdf()	Rice probability density function	62
rsdec()*	RS (Reed-Solomon) decoding	284
rsenc()*	RS (Reed-Solomon) encoding	284
set_CFO()	set up the CFO	368
set_parameter_11a	set the parameters for running "ieee_11a_CFO_est.mdl"	402
short_train_seq()	generates the short training sequence for 802.11a	364
sim_antipodal	simulates an antipodal (bipolar) signaling (in baseband)	125
sim_ASK_passband_coherent	simulates the coherent BASK in passband	176
sim_ASK_passband_noncoherent	simulates the non-coherent BASK in passband	177
sim_biorthogonal	simulates a bi-orthogonal signaling (in baseband)	135
sim_Delta_Sigma	simulates the delta-sigma ($\Delta\Sigma$) modulation	105
sim_DM	Delta modulation	101
sim_DPCM	Differential pulse code modulation	99
sim_DPSK_passband	simulates the QDPSK (differential QPSK) in passband	194

MATLAB routine name	Description	Page number
sim_DSB_AMSC	simulates the DSB-AM	74
sim_DSB_AMTC	simulates the conventional AM (DSB-AMTC)	77
sim_FM	simulates the FM (frequency modulation)	84
sim_FSK_passband_coherent	simulates the coherent FSK in passband	184
sim_FSK_passband_noncoherent	simulates the noncoherent FSK in passband	185
sim_MSK	simulates the passband MSK communication	214
sim_OQPSK	simulates the offset QPSK (OQPSK)	208
sim_orthogonal	simulates an orthogonal signaling (in baseband)	126
sim_PCM	Pulse code modulation	96
sim_PSK_passband	simulates the QPSK in passband	189
sim_QAM_passband	simulates the QAM in passband	199
sim_S_QDPSK	simulates the $\pi/4$ -shifted QPSK	210
sim_SSB_AM	simulates the USSB/LSSB (upper/lower single sideband)-AM	82
sim_TCM	simulates 8PSK with the trellis-coded modulation	296
slice()	slices a received signal according to the given constellation	222
source_coding()	performs the source coding on an input symbol sequence	259
source_decoding()	performs the source coding on an input coded sequence	259
sova()	Soft-output Viterbi algorithm for turbo coding	306
state_eq()	State equation used in conv_encoder()	286
TCM()	Trellis-coded modulation	329
TCM1()	TCM using TCM_encoder1() and TCM_decoder1()	334
TCM_decoder()	TCM decoder with a convolutional encoder touched by input	297
TCM_decoder1()	TCM decoder w. a convolutional encoder untouched by input	333
TCM_encoder()	TCM encoder with a convolutional encoder touched by input	296
TCM_encoder1()	TCM encoder w. a convolutional encoder untouched by input	332
TCM_state_eq1()	State equation used in TCM()	329
test_corr_circular	tests the MATLAB routine 'corr_circular()'	69
test_DSTBC_G2_PSK	applies the DSTBC with 3 transmit antennas for 4PSK	316
test_DSTBC_H4_PSK	applies the DSTBC with 4 transmit antennas for 4PSK	335
test_Rayleigh_fading	Rayleigh fading effect	60
test_unwrap	tests the function of unwrap()	84
trellis()	Trellis structure	304
tri()	Triangular pulse	9
tri_wave()	Triangular wave	4
turbo_code_demo	demonstrates turbo coding/decoding with logmap() or sova()	307
unwrap()*	undo the radian phase wrapped into $(-\pi, \pi]$	83-84
vector_quantization()	Vector quantization	104
vit_decoder()	Viterbi decoding	290, 291
vitdec()*	Viterbi decoding	292, 293
Viterbi_QAM	QAM with convolutional encoding and Viterbi decoding	326
xcorr()*	correlation	53
xcorr_my()	correlation	52
zfe()	zero-forcing equalizer	149

Index for Simulink Models

Simulink model name	Description	Page number
BCH_BPSK_sim	simulates BPSK with BCH coding	324
BPSK_squaring	simulates BPSK with squaring loop for carrier phase recovery	248
CFO_sim	simulates the CFO estimation and compensation	369
channel_estimation_sim	simulates the channel estimation for OFDM system	381
Delta_Sigma_sim	simulates the delta-sigma ($\Delta\Sigma$) modulation	105
do_OFDM0_sim	simulates a basic OFDM system	360
do_OFDM1_sim	simulates an OFDM system (IEEE Std 802.11a)	392
DS_SS_sim	simulates QAM with DS-SS(spread spectrum)	349
DS_SS2_sim	simulates QAM with 2-user DS-SS (CDMA) system	354
FSK_passband_sim	simulates the passband FSK (frequency-shift keying)	216
ieee_11a_CFO_est	simulates 802.11a system with CFO estimation/compensation	403-404
interleaving_sim	tries the interleaving and deinterleaving	384
MSK_passband_sim	simulates the passband MSK (minimum-shift keying)	223
PLL_sim	simulates a PLL (phase-locked loop) system	247
PN_generator_sim	tries generating the PN/Gold/Kasami sequences	342
PSK_carrier_phase_ timing_recovery	simulates PSK with carrier phase and timing recovery	251
PSK_passband_sim	simulates the passband PSK (phase-shift keying)	219
puncture_sim	tries puncturing and depuncturing	385
QAM_carrier_recovery	simulates QAM with decision-feedback carrier phase recovery	250
QAM_passband_sim	simulates the passband QAM	221
QPSK_Costas	simulates BPSK with Costas loop for carrier phase recovery	249
scrambling_sim	tries using Scrambler and Descrambler blocks	388
SRRC_filter	performs the cascaded square-root raised-cosine filtering	164
TCM_sim	simulates a Ungerboeck TCM (trellis-coded modulation)	330
Viterbi_QAM_sim	QAM with convolutional encoding and Viterbi decoding	327